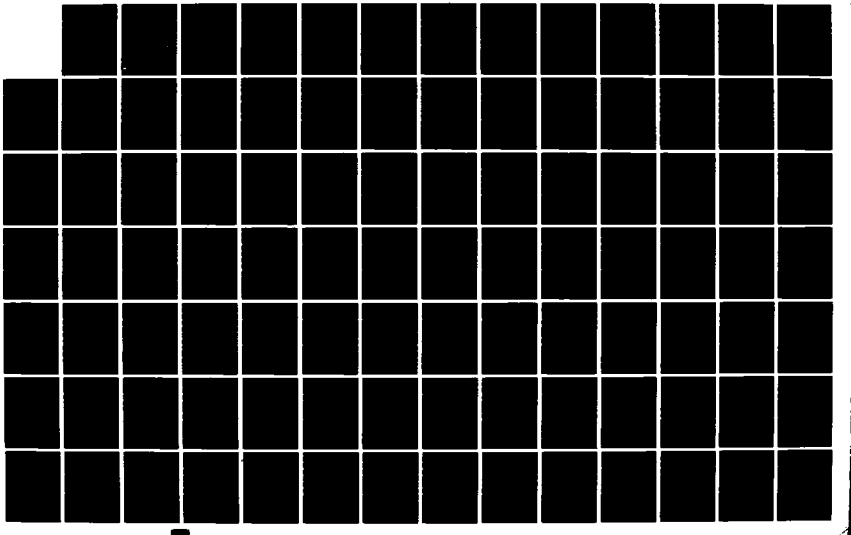
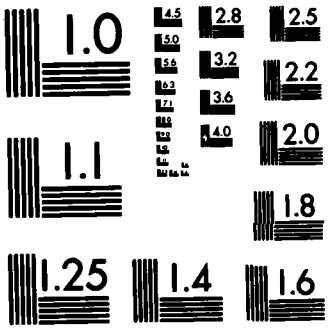
AD-A122 706 TACTICAL EXECUTIVE (TACEXEC): A REAL-TIME SECURE OPERATING SYSTEM FOR TACTICAL APPLICATIONS(U) SRI INTERNATIONAL MENLO PARK CA R J FEIERTAG ET AL. JUL 79 1/2

UNCLASSIFIED DAAB07-76-C-0368 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TACTICAL EXECUTIVE (TACEXEC): A REAL-TIME SECURE OPERATING SYSTEM FOR TACTICAL APPLICATIONS

Final Report

SRI Project 5545
Contract No. DAAB07-76-C-0368

July 1979

By: Richard J. Feiertag, Senior Computer Scientist
Karl N. Levitt, Program Manager
P. M. Melliar-Smith, Senior Computer Scientist

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Department of the Army
U.S. Army Electronics Command
Fort Monmouth, New Jersey 07703

Attention: David Egli, Project Monitor

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-1246

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: Final Report
2. GOVT ACCESSION NO.: AD-A122706
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): TACTICAL EXECUTIVE (TACEXEC): A Real-Time Secure Operating System for Tactical Applications
5. TYPE OF REPORT & PERIOD COVERED: Final Report
6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Richard Feiertag, Karl N. Levitt, and P.M. Melliar-Smith
8. CONTRACT OR GRANT NUMBER(s): DAAB07-76-C-0368

9. PERFORMING ORGANIZATION NAME AND ADDRESS: SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Project 5545

11. CONTROLLING OFFICE NAME AND ADDRESS:
12. REPORT DATE: ~~June~~ JULY 1979
13. NO. OF PAGES: 141
15. SECURITY CLASS. (of this report): Unclassified

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office):
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

real-time
multilevel security
tactical executive
computer operating system
formal specifications

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the design of a computer operating system (TACEXEC) that supports applications in a tactical military environment. The outstanding features of the system are that it is secure in the military multilevel sense and that it assures real time response to external events. TACEXEC is described both informally and in a formal mathematical notation. Formal and

DD FORM 1 JAN 73 1473
EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

informal definitions of security and real time performance are given and techniques for proving that TACEXEC meets these requirements are described. Several issues with regard to the implementation of TACEXEC are also discussed.

CONTENTS

ACKNOWLEDGMENTS

THE HIERARCHICAL DEVELOPMENT METHODOLOGY (HDM), one component of which is the language SPECIAL (SPECIfication and Assertion Language), used in this report to describe TACEXEC, was created primarily by Larry Robinson. Olivier Roubine, Bernard Mont-Reynaud, Robert Boyer, Brad Silverberg, Peter Neumann, and the authors of this report all contributed to the development of HDM. The development of HDM was sponsored at SRI primarily by the Naval Ocean Systems Center under Contract No. N00123-76-C-0195 (Lin Sutton, Project Monitor), by the U.S.Government under Contract No. DAAB03-73-C-1954 and the National Bureau of Standards under Contract No. <4430> and by the National Science Foundation under Grant No. DCR74-18661. In addition to the Army, under the current contract, the following organizations sponsored work at SRI on the multi-level security model: U. S. Government, Contract No. DAAB03-75-C-0399; ARPA Order 3341, subcontract with Ford Aerospace and Communications Corp. and Honeywell.

We are indebted to David Egli for the leadership and encouragement he offered us during the contract. During the early phases of the contract, Bernard Newman influenced the overall design of TACEXEC by alerting us to the particular needs of the Army for a secure, real time executive. Early discussions with Dennis Turner on primitives for process synchronization were extremely valuable.

I INTRODUCTION

This is the Final Report under Contract DAAB07-76-C-0368, entitled "Executive Software". The objective of this investigation was to design* a real-time operating system: The resulting system we produced is called Tactical Executive (TACEXEC). TACEXEC has been designed to have the following properties:

(1) Capable of handling the dispatching of real-time tasks, such as radar processing and weapons control.

(2) Adequate functional capability to support a wide range of application subsystems, for instance message handling.

(3) Efficiently implementable.

(4) Secure, in that separation of information according to a model of multilevel security is assured.

(5) Provable, in that formal reasoning it should be possible to show that critical properties of the system are satisfied. For TACEXEC, the critical properties are security and ensuring that tasks are dispatched according to their real-time needs.

(6) Portable, and capable of being implemented on any of a number of processors.

Important principles have emerged from the work of Dijkstra [6], Parnas [7], Hoare [8], and Floyd [9], the impact of which can be summarized as follows:

(1) It is possible to structure both a software system and the process of developing the system in such a way as to significantly enhance the reliability of tne system.

(2) It is possible to write formal specifications for a software system.

(3) In the near future it should be possible to formally prove the correctness of a system with respect to

---

* As will become clear later in the report, "design" is a collection of specifications (perhaps augmented by drawings, text etc.) from which an "implementation" (code that is directly executable on a processor) can be realized that will behave according to the specification.

specified properties. As mechanical verifiers become more powerful such a proof should be mechanizable.

In addition, the recent attempt to design numerous operating systems that are inherently structured has led to a collection of canonical designs that are, in principle, instantiable to meet many applications. Thus, the design of an operating system is not as onerous an undertaking as it once was.

The Computer Science Laboratory of SRI International, has developed a formal methodology for software development and verification called HDM (Hierarchical Development Methodology) [10] [11] [12] [13]. HDM is based on the ideas of Dijkstra, Parnas, Hoare, and Floyd, and on certain new principles that we developed in the course of applying HDM to real applications, e.g., TACEXEC. A key aspect of HDM is the language SPECIAL (SPECIfication and Assertion Language), which is used to specify the functional behavior of modules, the basic unit of composition in HDM. By composing a specification for a system, the effort of proving the correctness of the system can be decomposed into two steps: (1) proving the specification with respect to an abstract requirements statement, and (2) proving the design with respect to the specification. By hierarchically structuring the system, the implementation proof itself is decomposed into manageable units that mirror the system structure. This report presents a requirement statement that embodies the notion of multi-level security and contains sample proofs of the specifications with respect to that statement. We also present an informal proof demonstrating that the scheduler's specification implies sufficient computational capacity to dispatch tasks according to their real time needs.

The organization of the report is as follows. Chapter II summaries our approach toward the design of TACEXEC. A brief review of HDM, adequate for presenting the details of TACEXEC, is given in Chapter III. Chapter IV discusses the particular security requirements that TACEXEC is intended to satisfy. Chapter V, the most detailed in the report, presents the design of TACEXEC. Chapter VI presents a message processing system, an example of an application subsystem that can be

realized using TACEXEC. An introduction to some of the issues relevant to a future implementation of TACEXEC is given in Chapter VII. The criteria for selecting a high-level programming language that is matched to both HDM and TACEXEC are discussed in Chapter VIII. Finally, Chapter IX summarizes the impact of the work and lists topics that are extensions to the TACEXEC investigation. Six appendixes present:

- The specification of the modules of TACEXEC.
- The specifications of a module that provides message handling facilities.
- The rules for multilevel security.
- An illustration of how the TACEXEC design is proved with respect to multilevel security.
- A description of the algorithm that is used to schedule tasks on RTOS and a proof of its adequacy.
- A description of SPECIAL.

The design of TACEXEC is complete, and an implementation is planned at CORADCOM under the direction of Dr. E. Leiblein of CENTACS.

II DESIGN APPROACH

In this chapter we summarize the approach that was followed to attain each of the six objectives indicated in Chapter I. Before starting this discussion, let us review the well-established purposes of an operating system; namely to:

* Provide an interface (collection of operations) to the user that is more powerful than that associated with the bare hardware.
* Manage the resources of the computer, which generally means allocating them among the users so that the resources are kept as busy as possible.

Some difficulties are introduced by the need to handle real time behavior, but again there are well-established techniques for this. What makes this project challenging are the objectives of security, provability, portability, generality, and the desire of the Army to have an system that has sufficient generality, powere and simplicity to serve as a model for future systems development. Our approach toward achieving these goals is outlined below.

A. Real Time Behavior

The intent of TACEXEC is to satisfy the computational needs of tasks in a tactical environment. For real-time tasks this need is to process tasks within a specific time frame. Among the tasks that might be served by TACEXEC is a scanning radar, which delivers signals at regular intervals. Another task could be a fire control system that requires extensive service, but only in bursts. Message transmission is another task that is typically of low criticality, except that there might be a maximum delay that is acceptable for the transmission of a message. Each of these tasks poses different needs on TACEXEC, leading us to define three classes of tasks: iterative, demand, and background.

How does the system guarantee service requirements, particularly for the iterative and demand tasks? That is, how is it assured that the system loading is low enough such that the service needs will be met in the worst case of demand, but not too low so as to preclude the inclusion of additional tasks that could be handled. In Appendix E we consider this problem and demonstrate the following. There exists an optimal algorithm A (based on task deadlines) for the allocation and scheduling of tasks such that all tasks are dispatched on time and no other algorithm A' permits the allocation of additional tasks. Unfortunately, the processing time required for this scheduling algorithm probably precludes its use in typical real time systems. In its place we suggest a scheduling algorithm based on task priority -- an easily implemented algorithm -- for which tasks are guaranteed to be dispatched in time and the system can be loaded in excess of 60% of its processing capacity while guaranteeing that all tasks will be completed in time.

B. Functional Capability

In general, the interface provided by a real time operating system need not be as powerful as that for a general purpose time-sharing system (e.g., Multics). However, a real time operating system is intended to execute collections of interacting programs and should have sufficient functionality to realize some reasonably complicated subsystems. In TACEXEC we provide the following features at the user interface.

- virtual memory consisting of dynamically creatable address spaces and segments
- a file system
- a user I/O system
- processes
- synchronization primitives

Conspicuously absent from the system are: directories, linkage sections, and support for procedures, as well as other facilities that are found in general purpose operating systems such as Multics such facilities are

useful for program development and for setting up the real time system, but the hight overheads preclude their use in real time operations. It should be noted that these latter facilities could be built out of the TACEXEC facilities if desired. Thus, TACEXEC can be viewed as a kernel out of which a more complicated operating system could be constructed.

C. Efficiency

As indicated above, a classic principle underlying an operating system is the efficient management of resources (cpu, disk, main memory, etc.). In a real-time operating system this principle is in conflict with, and of secondary importance to the guaranteeing of timely service to tasks. In particular, the efficient management of tasks often introduces nondeterminism such that accurate performance prediction is not possible. Fortunately, the critical tasks (iterative and demand) typically require little memory and cpu time for each execution. Also, there is little sharing of I/O devices in a real time environment. Such tasks can therefore be given total access to all needed resources of the system for the short time required for execution. Also, if the program and data for these tasks is retained in main memory, then it is possible to guarantee (by the proof outlined in Appendix E) that the service needs of these tasks are met. This characteristic of the tasks led us to the decision that the virtual memory system is to be totally resident in main memory.

There are other issues regarding the efficient realization of TACEXEC. A high level language is desirable for easing the burden of implementation and to aid in portability (see F below). However, there are important features of a high level language that relate to efficiency. These are discussed in Chapter VIII.

Some operating system functions, such as interrupt handling and content switching, have very significatnt effects on the real tme performance of the system. As is discussed in Chapter VII, some additional hardware support for these functions would result in a substantial improvement in the performance of the operating system.

D. Security

TACEXEC is intended for an environment where multiple users have simultaneous access to the system, and each user wants to be assured that his information is not available to others without proper authorizations. That is, the system is not to be a vehicle for the erroneous disclosure of information. For this environment, the multi-level security model appear to be appropriate. In this model, each user has a clearance and a category set; the cartesian product of clearances and category sets define a partial ordering of security levels. The values for clearance are the conventional classifications: UNCLASSIFIED, CONFIDENTIAL, etc. The categories represent an orthogonal restriction, and include such "controls" as NATO, ATOMIC. The model requires that information stays at security level at which it originates or flows to move secure levels.

The model also includes the notion of integrity which provides additional restrictions on the flow of information. For example, using the security restriction alone, there are no limitations to the "upward" transmission of information. That is, the model does not prevent the "destruction" of a SECRET document by an UNCLASSIFIED user. The inclusion of integrity places limitations on such modification.

The model that includes security and integrity is developed in Chapter IV as an extension of the work by Bell and LaPadula, and Millen at Mitre. The model is also discussed in a recent paper [3] and is currently the basis for security proofs in the KSOS (Kernalized Secure Operating System) kernal [14].

E. Provability

TACEXEC has been designed to be provable, in particular by an automated program verifier. The main properties of concern are security and real-time adequace, i.e. the guaranteeing that tasks will receive promised service. Other properties, potentially of interest, have to do with to guaranteeing that the user interface operations provide the intended functional behavior.

As described in Chapter III, the development of systems according to HDM is accomplished in stages. For example, in the specification stage, each of the system modules (a module is provided for each "facility") is formally described by a specification. In the implementation stage, the operations of each module are implemented by a program. (The other stages are described later in the report.) A proof is associated with each stage. For example, it is possible to prove that the multi-level security model is satisfied by the specifications of the modules of the user interface of TACEXEC. Illustrations of such proofs, called design proofs are given in Appendix D. It is also possible to prove that the guarantee of service property is satisfied by the user interface specifications. Separate proofs, called implementation proofs, not yet carried out for TACEXEC, can demonstrate that the programs are correct with respect to the specifications.

This separation of proofs serves to simplify the overall proof process, as any useful decomposition of effort should do. In addition, it limits the amount of reproving that must be done as the system evolves. For example, a change to the implementation (eg. to enable TACEXEC to be installed on a different processor) does not require any change to the design proof if the specifications are left intact.

F. Portability

No real time operating system can be totally portable. In order to achieve efficiency, there will always be machine-specific code. Our concern was to design a system where the amount of effort required to move TACEXEC from a machine on which it is successfully executing to another machine is small.

Much of the effort involved in developing a system is associated with "design." In general, design is concerned with deciding what a system is to do, while avoiding details on how it is done. In HDM, the initial stages are concerned with design, while the later with realization. The output of these early stages is a set of specifications for the modules and a precise description of the

structure of the system. These can serve as the design for TACEXEC independent of the hardware on which it executes.

Furthermore, a system developed according to HDM is usually designed as a *hierarchy*. (The TACEXEC hierarchy consists of five levels.) Typically, the modules of the upper levels are implemented by software, the middle levels by a mixture of hardware firmware and software, and the lower levels by firmware and hardware. Thus, if a high level language is used for the implementation, many of the programs will remain intact in going from one hardware to another.

III SUMMARY OF HDM

In this chapter we summarize the Hierarchical Development Methodology (HDM) that was used in the development of TACEXEC. A reader familiar with HDM or another specification--oriented methodology may find it unnecessary to read this chapter.

A. Overview of HDM

In HDM a system is realized as a linear hierarchy (a sequence) of abstract machines, sometimes called levels. The top level is called the user-interface, as the user of the system perceives only this level. The bottom level is denoted as the primitive machine. These two machines together are denoted as the extreme machines. The remaining levels are called intermediate machines. An abstract machine consists of operations, each of which has a unique name and formal arguments. An operation is invoked (an invocation being similar to a subroutine call in a conventional programming language) by associating actual values with the operation's formal arguments. The invocation of an operation can return a value and/or modify the internal state (abbreviated as state) of the machines, as reflected by the values of the machine's abstract data structures. The return of an operation can be either a concrete value or an exceptional return, the latter corresponding to one of a number of conditions defined on the state of the machine and the supplied arguments.

The operations of the of the primitive machine are a mix of (1) a subset of those of the hardware on which the system runs, and (2) constructs of a programming language made available to the sytem developer to hide certain (usually tedious) features of the hardware.

The user-machine provides the operations that are available to the user of the system, and thus enriches the basic instructions of the

primitive machine. In selecting the intermediate machines, the designer is proposing building blocks to ease the step from the user-interface to the primitive machine.

A machine specification characterizes the value returned and the new state for each possible operatin invocation as dependent on the state of the machine.

In a hierarchy of machines M_1, ..., M_n (where M_1 is the primitive machine and the M_n the user-interface), the realization of M_i (i>1) is a two-step process. First, the abstract data structures of M_i are represented by those of M_{i-1}. Second, each of the operations of M_i is implemented as a program in terms of the operations of M_{i-1}. The collection of implementations for all machines M_i (i>1) constitutes the system implementation.

Usually, a machine is decomposed into smpler, separately implemented units called modules. A module, similar to an abstract machine, contains operations and data structures. For purposes of specification, the modules of a machine form a partial ordering. That is, for two modules m_i and m_j of a machine $m_i > m_j$ means that the specification of some operation of m_i depends on the values of data structures in m_j. For a well-conceived modular decomposition, there is little intermediate dependency.

Clearly, the system implementation is the desired end-product of the system development process. However, its creation is accomplished in stages as discussed below.

B. Stages of HDM

The creation of a system is organized into seven stages as follow.

(1) Conceptualization -- The statement of the intent of a system in abstract terms.

(2) Highest- and Lowest-Level (extreme) machine definition -- The description of the modular organization of the user interface and the primitive machine, and the declaration of the operations and data structures of each module.

(3) Intermediate machine definition -- The description of a sequence of abstract machines, between the two extreme machines, that serve as building blocks. Each intermediate machine is described in terms of modules and functions in a manner as employed in Stage 2 for the extreme machines.

(4) Module specification -- The act of giving a formal specification to each module of the system.

(5) Data representation -- The statement of how the data structures of each machine (except the primitive machine) are composed from the data structures of the next lower-level machine.

(6) Implementation -- The statement of how the operations of each module (except those of the primitive machine) are implemented in terms of those of lower-level modules.

(7) Coding -- The formulation of the implementations via constructs of an executable programming language. This stage can be avoided if the implementation of modules (Stage 6) is described using an executable programming language.

An important aspect of staged development is the separation of decisions. In each of the stages of development, the developer formulates decisions and writes them down in formal notation. The stages have been ordered so that the important decisions -- those that have major impact on the system--are confronted early in the development process.

It should be clear that a system is not necessarily created by a single pass through the stages. Significant iteration is certainly to be expected. The ordering of the stages might best be viewed as a scheme for making the decisions that ultimately lead to the system implementation and as a technique for documenting a completed system.

For TACEXEC, we have carried out the first four stages. A brief introduction to SPECIAL, the specification language appears in Appendix F.

IV SECURITY REQUIREMENTS FOR TACEXEC

In the TACEXEC we wish to enforce a restriction on the way information may be passed from task to task. The particular restriction of interest is called multilevel security. Each process has associated with it a CLEARANCE and a CATEGORY SET. The system has a fixed finite number of clearances that are totally ordered by the relation "less than". For example, the clearance CONFIDENTIAL is less than SECRET, which is less than TOP SECRET. For convenience, clearances are represented as integers.

A category set is any subset of the set of all possible categories. Examples of categories might be ATOMIC and NATO. The combination of a clearance and a category set is called a SECURITY LEVEL or equivalently ACCESS LEVEL; for simplicity, it is often called just a LEVEL when ambiguity is not likely to arise. A security level L1 is equal to a security level L2 if and only if the clearance of L1 is equal to the clearance of L2 and the category set of L1 is equal to the category set of L2. A security level L1 is said to be less than or equal to a security level L2 whenever the clearance of L1 is less than or equal to the clearance of L2, and the category set of L1 is a subset of the category set of L2. L1 is less than L2 whenever L1 is less than or equal to L2 and L1 is not equal to L2. Thus the set of all security levels can be partially ordered. Note that not all security levels are related by the partial ordering, e.g., two processes with respective security levels <SECRET, {ATOMIC}> and <SECRET, {NATO}> are not comparable. The security levels and the relation "less than" define a lattice since there is a minimum and maximum clearance, and a maximum set of categories.

In informal terms, a system is MULTILEVEL SECURE if and only if, for any two processes P1 and P2, unless the security level of P1 is less

than or equal to the security level of P2, there is nothing that P1 can do to affect, in any way, the operation of P2. That is, P2 is not able to know anything about P1, not even the existence of P1. This constraint implies that P1 cannot affect the operation of P2 using an intermediate process P3. It is not possible for a process at a higher level to transmit information to a process at a lower level. Therefore, INFORMATION CAN ONLY FLOW UPWARD IN SECURITY OR REMAIN AT THE SAME LEVEL, i.e, can only flow to processes of greater or equal security level.

The above constraint is consistent with the real military security situation, since -- for example -- an individual whose category set contains only ATOMIC cannot pass information to an individual whose category set does not contain ATOMIC, independent of the latter's clearance or the other components of his category set.

A. Manifestation of multilevel security in the TACEXEC design

In order to ensure that the TACEXEC design is multilevel secure, the rule of upward information flow must be manifested in the specifications for the TACEXEC. Multilevel security appears in the specifications in three ways. First, each repository of information in the specifications must be assigned a security level. This is accomplished by adding one additional argument to each primitive V-function (primitive V-functions are the data structures of a machine and thus are repositories for information) which gives the security level of the particular reference to that primitive V-function. Second, an argument is added to each visible function of the specifications that gives the security level of the invocation of the function. This argument is implicit in that it is supplied by the system, not the calling procedure, thereby guaranteeing its accurracy. Third, exceptions are included in each function that abort any function invocation that would cause information to be transferred from one primitive V-function reference to another primitive V-function reference or from the caller of the function to or from a primitive V-function reference in a manner inconsistent with the upward flow of information.

Using this technique, multilevel security can be added to any design, however the added exceptions may be so restrictive as to make the resulting system useless or very difficult to implement. It is therefore advisable to incorporate multilevel security into the design as it is being formulated.

B. Multilevel Integrity

The multilevel security model does not prohibit a process at some security level from modifying information at a higher security level. However, there are many cases in which such a prohibition is desirable. Biba [2] has identified the concept of integrity to solve this problem. Integrity is the precise formal dual of multilevel security. In addition to a security level, each process of the system has an associated integrity level. The set of integrity levels is identical to the set of security levels and has the same relation "less than". A system has multilevel integrity if and only if, for any two processes P1 and P2, unless the integrity level of P1 is greater than or equal to the integrity level of P2, there is nothing that P1 can do to affect, in any way, the operation of P2. Therefore, information can only flow downward in integrity or remain at the same integrity level. Integrity can be used to limit the upward flow of information enforced by multilevel security. It is important to remember that a process's security level and its integrity level need not be the same. The primary advantage of using integrity as a further means of restricting information flow is that, being the formal dual or security, it adds no significant complexity to the security model and no significant complexity to the proof of a secure system design.

Multilevel integrity has been included in the TACEXEC design of Appendix A. Since anything that applies to mutlilevel security also applies to multilevel integrity as its dual, there is no further discussion of integrity and the reader should understand that all discussion of multilevel security applies to multilevel integrity as well.

C. Proof of the Multilevel Security of the TACEXEC design

In order to prove that the TACEXEC design is multilevel secure, it is necessary to have a precise statement of the multilevel security described above and a precise formulation of the TACEXEC design, and to prove that the precise statement of security and the formulation of the design are consistent.

Several precise statements of multilevel security have appeared in the literature including [1] and [5]. The precise statement of multilevel security to be used in this report is given in Appendix C. The precise formulation of the TACEXEC design is given as the specifications of Appendix A. The proof of consistency is demonstrated in Appendix D.

D. Security of the Implementation

Multilevel security is the only explicit form of security specified for the TACEXEC. However, when implementing the TACEXEC an additional form of security is necessary that is implied but not explicitly stated in the specifications. It is essential that the implementation of the algorithms that realize the specifications be inviolate. In other words, it must not be possible for an TACEXEC process to be able to modify the software or hardware that implements the specifications. For hardware, this security constraint is generally equivalent to the physical security of the machine. For software (and firmware) the problem is more difficult. Many modern machines have means of protecting system programs from being tampered with by applications programs. Features such as separate user and supervisor states allow such protection to be implemented. However, guaranteeing the isolation of the implementation software is essential to any other forms of security provided by the system.

V SYSTEM DESIGN

The TACEXEC provides four basic resources to allow users to perform desired computations in real time. The four resources are:

PROCESSES - A process is the entity that executes programs and, therefore, performs the desired computations.

VIRTUAL MEMORIES - Associated with each process is a virtual memory. The virtual memory consists of a fixed number of segments. Each segment is a linear address space of fixed size memory units (words, bytes, bits, etc.). All the memory units are directly addressable as operands to instructions and the contents of a memory unit can be accessed in an amount of time on the order of an instruction execution time of the machine. There is a limited amount of virtual memory storage.

FILES - Files provide an alternative form of storage to the virtual memory. Files may only be accessed in their entirety, i.e., the entire contents of a file may be copied into a segment of a virtual memory or the contents of a segment may be placed in a file. It is not possible to access individual words, bytes, or bits of a file. The amount of file storage will, in general, be much greater than the amount of virtual memory storage.

I/O CHANNELS - Channels provides a means and minimal conventions for communicating with particular I/O devices that are connected to the TACEXEC. Means are provided for sending data and commands to devices and for receiving data and status information. Some basic synchronization is possible between devices and processes.

These resources are sufficient for accomplishing any necessary computation. They have been chosen to permit simplicity of system design, ease of use by programmers, minimal system overhead, and predictable real time performance.

A. DISPATCHER

The DISPATCHER creates the abstract object called a process. A process is the entity that executes the instructions of a program necessary to compute some task. The DISPATCHER provides three operations for instantiating a process (i.e., creating a process that will execute the instructions of a given task), SCHEDULE_ITERATIVE_PROCESS, SCHEDULE_DEMAND_PROCESS, and SCHEDULE_BACKGROUND_PROCESS. These three operations require, as arguments, sufficient information to locate and interpret the instructions to be executed, and to assure the proper real time behavior of the process. The arguments common to all three operations are:

p - the identifier of the process

p_c - the program counter which is the address of the next instruction to be executed by the process p

m_s - the contents of the machine registers of the process when it begins executing

The remaining arguments describe the desired real time behavior of iterative and demand processes and are different for each:

ITERATIVE PROCESSES

int - the interval of periodicity for the process (the process must be permitted to run once in every "int" time units

dur - the duration of execution of the process (the process must be allowed to run for "dur" time units within the given interval)

begin_time - the time at which the process should first begin running

DEMAND PROCESSES

e - the external event whose occurrence will cause the process to begin running

dur - the maximum number of time units the process will run when an event "e" occurs

min_period - the minimum number of time units between two occurrences of the event "e"

Whenever a process wants to wait for some notification from another process, or has completed its current iteration (in the case of an

iterative process), or has finished processing an occurrence of an event (in the case of a demand process), it does so by invoking the operation BLOCK. The operation block will not return to the invoking process until the awaited notify occurs, the time for the next iteration of the process arrives, or the awaited event occurs. Although a process cannot detect that processor multiplexing is taking place, an invocation of the operation BLOCK is an indication to the DISPATCHER that the invoking process can relinquish the processor on which it is currently running and that a different process can now use the processor. However, an invocation of BLOCK looks to the invoking process simply like a delay.

The operation TICK describes the effect of the passage of time (i.e., the incrementing of the clock). It is not necessary (and, in fact, would not be reasonable) to implement the indicated effects at the time the clock is incremented; these effects can be achieved by other means.

The operation OCCURRENCE indicates that the event given as its argument has occurred causing demand processes awaiting that event to begin running. The operation NOTIFY causes all processes waiting for the given "wakeup" to begin running. A process invoking the operation WAIT, with some "wakeup" as an argument, indicates that when that process next invokes BLOCK with an argument of TRUE, that the process will await the occurence of a NOTIFY on that wakeup. The operation CONTINUE cancels the effect of the preceding WAIT if one has been invoked since the preceding invocation of BLOCK. The remaining operations of the DISPATCHER: CREATE_PROCESS_IDENTIFIER, CREATE_EVENT, and CREATE_WAKEUP, simply perform the action indicated by their name.

In order to implement the above operations, any implementation of the DISPATCHER must maintain state information. There must be a list of all currently instantiated processes and information about the state of execution and the scheduling of these processes. This information is represented in the specification of the DIPATCHER by the primitive V-functions H_PROCESS_EXISTS and H_PROCESS_INFO. The information about each process maintained by the V-function H_PROCESS_INFO consists of the

type of the process (i.e., iterative, demand, or background), the next time the process should begin to run, the interval of iteration for iterative processes or the minimum time between events for demand processes, the run time required by an iterative or demand process for each iteration or event occurrence, the event a demand process should wait for, the wakeup a process is waiting for if it is waiting for a wakeup, the program counter of the process, and the state of the process' registers. Each process can be properly scheduled and dispatched based on this information. The remaining piece of information returned by H_PROCESS_INFO, processing_remaining, is the number of time units of duration that have not yet been expended in the current iteration. This information is not essential to the scheduling algorithm, but it does increase the reliability of the system. If a process runs for longer than its stated duration during some iteration, then it is permitted to continue running on a low priority basis, i.e. it may be preempted by a higher priority process. In this case the errant process may miss some iterations, but it will not effect the real time performance of any other processes. In addition, the DISPATCHER maintains a list of processes waiting for a wakeup (H_WAITING_PROCS), the current time (H_TIME), and existing events and wakeups (H_EVENT_EXISTS and H_WAKEUP_EXISTS).

Most of the complexity of the specification of the DISPATCHER is in the definitions. It is, therefore, useful to briefly describe the purpose of each of the definitions. The definition TIME_CRITICAL_PROCESS returns TRUE if its argument is an iterative or demand process. PROCESS_READY returns true if its argument could be running but isn't. PROC_PRIORITY returns the priority of a given process. Iterative and demand processes have higher priority than background processes. Also, processes with shorter intervals have higher priorities than processes with longer intervals. READY_PROCESSES is a list of processes ready to run in decreasing order of priority. PREEMPTABLE_PROCESSORS is a list of all the processors in increasing order of the priorities of the processes running on each of the processors. The PROCESSOR_UTILIZATION of a given iterative or demand

process is simply the fraction of the processing power of a single processor that the process could possibly consume in the worst case. For an iterative process, the worst case occurs if the process runs for its full duration at each iteration. For a demand process, the worst case occurs when events occur at the maximum possible rate and the process runs for its full duration at each occurrence. LN_2 simply represents the natural logarithm of 2.

Once these definitions are understood, the effects of each of the specified operations are simple to understand. For example, consider the specification for the BLOCK operation. The effects of this operation state that the first (higest priority) process on the list of ready process will begin to run on the processor previously occupied by the process invoking BLOCK. The time for the next iteration of the newly running process is revised and the processing remaining for this process is set to its full duration. The specification of the time of the next iteration (next_service) appears to be rather complex. This complexity is necessary for the case, discussed earlier, in which a process runs for a time exceeding its stated duration. In this case, it will miss all iterations which arrive until it finally calls block. This somewhat complex specification for next_service permits the process to become resynchronized with its iterations.

Or consider the operation TICK. If any ready process has a higher priority than some running process, then the running process is preempted by the ready process and the high priority ready process runs. The real time clock is incremented and the processing time remaining for each running process is decremented. Note that the processing time for all processes that are waiting and would have been runnning if they had not been waiting are also decremented. Therefore, a waiting process is considered to be the same as a running process for the purpose of computing its real time performance.

B. SYSTEM INPUT/OUTPUT

The TACEXEC does not support any particular types of input or output device, it simply provides a means for communicating with any device that may be connected to the system. The communication protocol is made very simple and general to make it easy to use and to permit utilizing a great variety of different types of devices. Since the opertions of the SYSTEM INPUT/OUTPUT module are not visible at the TACEXEC system interface, no security constraints are placed on the use of the various communication channels. The multilevel security constraints are imposed by the USER INPUT/OUTPUT module which is the user interface to the communication channels.

Each communication channel is identified with an integer. There are four operations for communicating with an input/output device. Each of these operations requires an integer as an argument to identify the channel being addressed. This integer is called the "device index". The READ_DEVICE operation inputs a unit of data from the device. WRITE_DEVICE outputs a unit of data to the device. SEND_COMMAND sends control information to the device and RECEIVE_STATUS returns information about the status of the device.

The information transferred by each of these operations is bufferred. Each channel has four buffers, one each for input data (H_INPUT), output data (H_OUTPUT), commands (H_COMMAND), and status information (H_STATUS). This means that the process communicating with the device and the device itself do not have to be completely synchronized. However, some synchronization is necessary. For example, a process cannot send a second unit of data to a device until the first unit of data has been received by the device. In general, if a process tries to send or receive information to or from a device at a speed significantly faster or slower than the device is receiving or sending that information either an exception will result or information may be lost.

In order to send or receive information to or from the system, the device has four operations that it may invoke. DEVICE_OUTPUT receives

data sent to it by the system and DEVICE_INPUT sends data to the system. DEVICE_COMMAND receives control information sent by the system and CHANGE_STATUS modifies the status information returned by RECEIVE_STATUS. In addition, CHANGE_STATUS may cause an event to occur if its second argument is TRUE. Each device can cause some particular event to occur and the event caused by each device is determined by the operation SET_EVENT. Note that the device index of the operations that can be invoked by a device is an implicit argument of the operation. This is because the system knows which device index corresponds to each channel and it is not necessary and undesirable for the device to provide this information. The four operations that can be invoked by devices would normally be implemented in hardware. The operations invoked by TACEXEC processes may or may not be implemented in hardware.

C. VIRTUAL MEMORY

The virtual memory of the TACEXEC consists of some address spaces each of which contains some number of segments. Each process in the TACEXEC is associated with some address space. The address space contains all the directly addressable storage accessable to the process. Each address space can contain some fixed predetermined number of segments. A segment is essentially an array of storage units (words, bytes ,bits, etc.). A process can address a storage unit by giving a segment number and a segment index. The segment number identifies a segment in the address space associated with the process, and the segment index specifies a storage unit in that segment. When a process is executing on a processor, all the segments of the address space associated with that process will be contained in directly addressable memory. This assures that data in storage can be addressed quickly in order that real time constraints can be met.

Each address space has an associated storage quota. The storage quota of an address space is the maximum number of storage units that can be contained in segments belonging to that address space. Both address spaces and segments can be created and destroyed. A segment can

be shared (accessed) by more that one address space, however, each segment is owned by only one address space, the address space in which the segment was created, and the storage units of the segment are considered to be part of the quota of the owning address space.

The VIRTUAL MEMORY module is the lowest level module containing operations accessable at the system interface. Therefore, it is necessary that these operations be multilevel secure. Much of the complexity of the VIRTUAL MEMORY module is due to the necessity to enforce the multilevel security constraints. In the following discussion the virtual memory is discussed without mention of multilevel security, as the basic design is not a consequence of any security constraints. The addition of the multilevel security constraints to the basic virtual memory design is straghtforward.

The data necessary to support the virtual memory includes the currently existing address spaces (H_AS_EXISTS), the quota for each address space (H_AS_SIZE), the segment corresponding to each segment number of each address space and whether or not that segment is owned by the address space (H_AS_ENTRY and H_AS_ENTRY_OWNED), and the contents of each storage unit of a segment (H_READ). The primitive V-functions H_AS_USED and H_SEG_USED are needed to generate designators for address spaces and segments.

The function CREATE_ADDRESS_SPACE returns a designator for a newly existing address space with the given quota and at the given access level. Initially a newly created address space contains no segments. A segment can be created by CREATE_SEGMENT. The newly created segment will have the given segment number in the given address space. The size of the segment and its initial contents are also given in the invocation of CREATE_SEGMENT. The address space in which a segment is created is said to "own" that segment. This means that the storage used by the segment is charged against the storage quota of that address space and that the segment may be deleted only from that address space. The function GET_SEGMENT adds an already existing segment to the given address space. This function is used to allow processes operating in

different address spaces to share the same segment. The function DELETE_SEGMENT removes a segment from the given address space. If the segment being removed from the given address space is owned by that address space, then the segment is deleted. Deleting a segment effectively removes the segment from all address spaces from which it could be accessed. The function DELETE_ADDRESS_SPACE deletes the given address space and all segments owned by the given address space.

The functions SEGMENT_READ and SEGMENT_WRITE read and modify respectively the specified word in the segment with the given segment number in the address space of the invoking process. The specifications for the function SEGMENT_WRITE are somewhat complex because the system permits information to be written by a process of lower security level to a segment of higher security level. However, in this case the system is not permitted to inform the user whether or not the writing operation was actually accomplished nor even if the segment being written actually exists. Therefore, if the function GET_SEGMENT is invoked to enter a segment from a higher level address space into the invoking process's address space, it is necessary for the system to pretend that the segment exists even though it may not. One of these phantom segments is represented in the specifications by having H_AS_ENTRY for the segment being undefined and having H_AS_OWNED being defined.

The functions SEGMENT_CREATE and SEGMENT_DELETE are nearly identical to the functions CREATE_SEGMENT and DELETE_SEGMENT. SEGMENT_CREATE and SEGMENT_DELETE are intended for use at the system interface and therefore, are restrictive in how they may be invoked. CREATE_SEGMENT and DELETE_SEGMENT are intended for use within the system and therefore, permit greater flexibility.

D. FILE SYSTEM

The file system contains data which is collected into repositories called files. Unlike segments in the virtual memory, the data in files may not be accessed individually by words, the file may be accessed only as a unit. In order to access individual elements of data in a file,

the contents of the file must be loaded into a segment where the data may then be accessed using the virtual memory operations SEGMENT_READ and SEGMENT_WRITE. It is intended that the data in files be stored on bulk storage devices for these devices are well suited to storage accessed in this manner.

As with virtual memories there are storage quotas on the amount of data that may be stored. There is a storage quota for each access level that restricts the number of data storage units that may be collectively contained in files at the access level.

All the state information of the file system is embodied in the V-function H_FILE_CONTENTS of the specifications. Each file has a name and an access level. If the value of H_FILE_CONTENTS is undefined for a given name and access level, then that file does not exist. If, however, the value of H_FILE_CONTENTS is defined then the value is a vector of the data in the file.

The operations CREATE_FILE and DELETE_FILE create and delete respectively a file of the given name at the access level of the invoking process. A process cannot, of course, create a file with the same name as an already existing file at that level. The operation LOAD_FILE copies the contents of the file with the given name into a newly created segment with the given segment number in the invoking process's address space. The access level of the file is determined by a given access level argument to LOAD_FILE. The operation UNLOAD_FILE copies the contents of the segment with the given segment number in the invoking process's address space to the file with the given name at the access level determined by the given level argument. The file must already exist and the previous contents of the file are destroyed. The operation APPEND_TO_FILE is similar to UNLOAD_FILE except that the previous contents of the file are not destroyed and instead the data from the segment is appended to the end of the file.

E. PROCESS PRIMITIVES

The operations of this module are much the same, in function, as those of the DISPATCHER. However, at this level, the level of PROCESS PRIMITIVES, the specifications are written from the point of view of an individual process, rather than from the point of view of the system. For example, in the dispatcher, an invocation of the operation BLOCK is specified as one that performs processor multiplexing, whereas in the process primitives, an invocation of the operation BLOCK is viewed as a simple delay since this module specifies the point of view of only a single process. In an implementation, both specifications of BLOCK would be identical. The two different views of these modules are both useful, depending on the particular aspect of the operation of the system in which the reader is interested.

The operation CREATE_EVENT returns a designator for a new event. The argument to CREATE_EVENT gives the minimum period of the event, i.e., the minimum time between two occurrences of the event. The operation occurrence indicates that the given event has occurred. Any demand process awaiting the occurrence of the given event will be activated.

An invocation of the operation BLOCK causes the invoking process to be delayed. If the invoking process is an interative process then the process will be reativated when time arrives for the process' next iteration. If the process is a demand process then BLOCK will return when the event associated with the process next occurs.

The operation TICK indicates the passage of time. Each invocation of TICK indicates the passage of a single time unit. In an implementation, TICK will most likely be invoked by and implemented by special clock hardware.

The operations CREATE_ITERATIVE_PROCESS, CREATE_DEMAND_PROCESS, and CREATE_BACKGROUND_PROCESS create respectively iterative, demand, and background processes providing the appropriate parameters for establishing the new processes. Note that each of these operations take

an address space as an argument. This address space is the one that will be associated with the newly created process. One of the exceptions to each of these operations is RESOUCE_ERROR. This exception will occur if the system is so heavily loaded with iterative and demand processes that the addition of the new process may not allow the system to meet the real time scheduling constraints given in the specifications. Note that since these operations are to be implemented in terms of the operations SCHEDULE_ITERATIVE_PROCESS, SCHEDULE_DEMAND_PROCESS, and SCHEDULE_BACKGROUND_PROCESS of the DISPATCHER, the exception RESOURCE_ERROR must be true whenever the addition of the new process to the system would violate the assertions concerning overloading given in the specifications for SCHEDULE_ITERATIVE_PROCESS and SCHEDULE_DEMAND_PROCESS.

The use of the exception RESOURCE_ERROR in the operations CREATE_ITERATIVE_PROCESS, CREATE_DEMAND_PROCESS, and CREATE_BACKGROUND_PROCESS, could produce a multilevel security violation. This is because RESOURCE_ERROR does not precisely specify the conditions under which the exception occurs. It is possible that in some implementation, the occurrence of this exception is dependent upon data at various security levels and, therefore, may compromise highly secure information. In this case the potential security violation is deliberate. The processing resource is to be shared by all processes at all security levels. This means that it is possible that if the machine is loaded with TOP SECRET processes, an UNCLASSIFIED process may not be able to be created (this is technically a security violation). Avoiding this security violation would require strict partitioning of processor resources and would lead to less effective processor utilization. In a real time system, highly effective processor utilization is important and so this technical security violation was felt to be acceptable. In practice, administrative controls on adding tasks to the system would eliminate the security problem (the violation would create only a noisy and very low bandwidth information channel in any case).

The operation SEGMENT_GET is simply a reformulation of the operation GET_SEGMENT of the VIRTUAL_MEMORY in terms of the destination process rather than the destination address space. This is possible since the association between process and address space is known at the PROCESS PRIMITIVES level whereas it was not known at the VIRUAL MEMORY level.

The information necessary to define these operations includes the primitive V-functions H_PROCESS_EXISTS and H_PROCESS_INFO which indicate respectively whether or not a given process exists and what its scheduling parameters are. H_EVENT_MIN_PERIOD indicates whether or not a given event exists and what its minimum period is. H_TIME gives the current time.

F. USER INPUT/OUTPUT

User input and output is very similar to system input and output, the only difference being that since the user input and output is visible at the system interface it must obey the constraints of multilevel security. Therefore, each device is assigned an access level (by the functional parameter H_DEVICE_AL) and each operation includes an exception that is true if the invoking process is not at the proper level for performing the operation upon the given device.

G. PROCESS COORDINATION

The two operations of the PROCESS COORDINATION module provide a means of coordinating the the activities of two or more cooperating processes. The operations are the standard P and V operations and they use the value of a word in a segment as the value of a semaphore. The V operation increments by one the value of the given word. The P operation decrements by one the value of the given word, however P will not cause the value of the semaphore to become negative, it will wait until it can safely decrement the value of the word without making it negative.

VI AN APPLICATION SUBSYSTEM--MESSAGE PROCESSING

In this chapter, we develop a subsystem that can be realized in terms of the operations of the TACEXEC user interface. Two purposes are served by this exercise:

* We demonstrate the general utility of the TACEXEC user interface.
* We present a module for handling messages in a multi-level secure manner--an interesting module in its own right.

In the following sections we present:

* A description of the specifications of the message system.
* Some embellishments that the reader might consider for subsequent incorporation.
* A brief discussion of the realization.

A. MESSAGE SYSTEM module

We hypothesize a new abstract machine above the TACEXEC interface, consisting of the modules MESSAGE SYSTEM and VIRTUAL MEMORY. The specification of the latter was previously discussed so we concentrate here on the specification of the MESSAGE SYSTEM module which is depicted in Appendix B. It is convenient to view the module as being composed of two groups of functions: (1) those associated with "user" management, and (2) those associated with the handling of messages.

The former group consists of the functions: USER_EXISTS, USER_EVER_EXISTED, CREATE_USER, and DELETE_USER, for which the major design decisions are the following:

* Only a particular user, the "security officer", can cause the creation or deletion of users. The security officer exists at all access levels.
* Associated with each user is a designator (USER_ID) and a name. The former acts as a password, presumably available only to the security officer and the user. The "name" is

known to all other users who wish to communicate with the user in question.

* A user when created at some access level al, also exists (can send or receive messages) at all access levels not exceeding al.
* For each access level there is a quota of potential users, as declared by the parameter MAX_USERS.
* There is no reuse of USER_ID designators. The function USER_EVER_EXISTED records all of such designators that have ever been associated with users.
* User names can be recycled, but all users in existence at any instant have unique names.
* The security officer is not required to obey the multi-level security rules. For example, the CREATE_USER function invoked at access_level al, apparently affects the V-function USER_EXISTS at all levels below al.

Now let us consider the design decisions associated with the functions: MSG_CONTENTS, SND_MSG, READ_MSG, and DELETE_MSG, for which the major design decisions are the following:

* A message is a vector of words.
* All users at a level have a quota on the number of messages that can be received and on the total amount of memory that can be expended for message storage.
* A user sending a message to another named user must identify the access level at which that user will read the message.
* A sent message is the contents of a segment.
* A user identifies a message by number for the purpose of reading or deleting it.
* The handling of messages obeys the rules of multi-level security and integrity. Thus, if the transmitting of a message by a user operating at security level n to a user at n1, n1 > n, would precipitate an exception due to exceeding the quota of the user at n1, this condition is not made apparent to the user at n; the message is simply not transmitted.
* The system determines if a user attempting to use the message system is a valid user.

B. Embellishments to the Message System

Since our intention was primarily to illustrate an application system for TACEXEC, rather than construct a elegant message handling system, we have left out many desirable features. (A good example of a powerful message is that developed for the ARPANET.) Among the features that could be added to our system are the following:

* Provide quotas on the basis of access levels rather than just users. Clearly, system-wide quotas are not acceeptable in a multi-level security environment.
* Allow a user operating at a level n to read all messages destined for him at any level n2 <= n, rather than requiring him to read only those messages at his operating level.
* Allow for the forwarding of messages..
* Allow the copying of messages into segments.
* Provide a realistic LOGIN function as part of a command level module.

C. Realization of Message System

As indicated previously, the operations of the TACEXEC user interface are to be used to realize the MESSAGE SYSTEM module. Below, we sketch some of the representation and implementation decisions that could form the basis for such a realization.

* A separate address space would be established to provide storage for the MESSAGE SYSTEM. In particular, a segment would store the global information on users. Some of this information could be kept on a file, with mimimal information retained on a segment.
* Each mail box (characterized by the function MSG_CONTENTS) would be represented by a segment.
* The USER_ID designator type could be represented by a set of characters.
* It is assumed that message handling is not necessarily a critical task. Hence, the act of sending, reading, or deleting a message could be handled by a background process that is created just for that task. If for some applications, message handling is critical, then a demand process would be created at system initialization time to handle messages.

VII TOWARDS THE EFFICIENT IMPLEMENTATION OF TACEXEC

We may regard the operating system as providing a virtual machine, or rather several virtual machines, on which are run the real time applications programs. This virtual machine could be implemented entirely by software interpretation, as for instance an APL virtual machine is, but this would be extremely inefficient. An implementation entirely in hardware might be efficient, but would be expensive and would take a long time to develop. Consequently, we must consider implementations in which the operating system virtual machine is implemented in part by software, and in part directly by hardware. In particular, we must consider implementations using the kinds of processor hardware that is readily available, for instance, the PDP11 architecture and the GYK-12. Examples of the operations that must be implemented in software are:

create_segment,
load file,
create_demand_process,
schedule_iterative_process,
etc.

Such operations must be implemented as supervisor calls to operating system software procedures with security privileges. Examples of operations that can be implemented directly by hardware, with appropriate safeguards, are:

simple arithmetic operations,
conditional brands,
iteratior,
procedure entry and exit,
etc.

Some operations might be implemented as supervisor calls to software procedures, but are used sufficiently frequently and are sufficiently

simple that, for an efficient system, consideration must be given to providing these operations by a short sequence of hardware operations. An obvious example for the PDP11 architecture is the user input/output operations.

Clearly the operations implemented in software will be slower, possibly much slower. Much of this delay is caused not be the inherent complexity of the operation but instead by the need to change protection context so that the operating system procedure can access security sensitive data that must not be available to real time application programs. Another contribution to the cost of software operations is the very careful checking of parameters to the operation, which is necessary to ensure that accidental or malicious use of the operations cannot subvert the security of the system.

Consequently, we must expect that a software implemented operation will take at least 100 times as long as a operation implemented directly by hardware, and there are many examples of existing systems where this factor is 500 or more. This kind of large speed ratio between hardware and software implemented operations forces us to consider the number of software operations that will be needed. If the ratio is 100:1 and 1% of the operations are software implemented, then the software implemented operations will need 50% of the processing time of the system. Now typical real time application programs are very short, seldom requiring more than 100 operations per activation, of which at least two and probably four to six will be software implemented. Further, to keep the ratio as small as 100:1 will not be easy. Thus the proportion of the processing time needed for software operations could rise well above 50%.

It is important not to regard this software "overhead" as wasted time. These operations implemented in software are important, necessary parts of the total system. They are no more wasted than any other necessary operations, but they are expensive and thus their implementation and their use must consider this expense.

In a non-secure operating system, many of the operations will (or at least could) be equivalent to those of TACEXEC, and may be as expensive as those of TACEXEC. But where real time performance is important a non-secure operating system can be designed so that the time required for software implemented operations is substantially less than in TACEXEC. This saving would be obtained by eliminating the parameter checking and changes in protection described above, and by allowing every program access to all the data of the system. An example of such an approach is the stack-based interrupt handling of the smaller PDP11. Not only does this approach violate the security requirement, but it also prejudices the reliability of the system for a fault in any one program could damage other programs or the operating system. A fully secure system should not be significantly more expensive than a system capable of preventing accidental damage. Below we consider alternative methods of improving the real time performance of TACEXEC without loss of security.

The design of TACEXEC, like almost all other modern operating systems, is designed so that the simple arithmetic, logical, and flow of control operations can be implemented by hardware without software intervention. Almost all the checking for security level and category is confined to the segment_create and get_segment operations, which include a segment within the address space of a process. Subsequently the checks, that reading is only from segments within that address space, can be performed by hardware protection mechanism. The definition of segment_write in TACEXEC is appropriate only if the hardware protection mechanism has a write only setting for access to segments (the PDP11 does not). The value of this feature is uncertain and in the absence of a hardware write only capability the specifications could be simplified.

TACEXEC does not specify, but does not preclude, an implementation in which segments are split into fixed size pages for storage allocation, nor does TACEXEC specify or preclude the movement of segments or pages between storage mechanisms of different speeds. An

implementation of this kind would certainly be more complex and less certain in its real time capabilities, but the security properties of the system would not be substantially affected.

Similarly TACEXEC does not provide for input-output operations that transfer multiple words between storage and peripheral. Within the PDP11 architecture it is possible to set the hardware protection mechanisms to allow direct transfers of single data words between processor and peripheral, without intervention by operating system software and without prejudice to the security of the system. Devices which require autonomous transfer of many words of data cannot safely be used directly by real time application programs, but must be managed by secure operating system procedures, with correspondingly higher overhead. It would be possible to extend TACEXEC to include these autonomous transfers, though possibly extensions for specific devices, e.g., communications, displays, etc, would be as appropriate.

Inclusion of both paging and user defined autonomous transfers significantly increases the complexity of the design, indeed removes the design from the area of small operating systems into that for large mainframe operating systems. Security breaches have been found in many existing operating systems because of the interactions between these two features.

It is clear that the parameter checking is essential for security and that performance improvements cannot be obtained by skimping on these checks. Thus performance improvements must come from either reducing the context switching time or by increasing the speed at which operating system procedures are executed. The first of these, reducing the context switching time, might be achieved by providing a privileged supervisor mode with a second set of registers in which all time critical operating system procedures are obeyed. The second alternative, increasing the speed of execution of operating system procedures, could be obtained by implementing them in horizontal microprogram. This second alternative automatically includes the first, for microprograms regularly have access to many additional registers,

and it also reduces the time required for parameter checking and other operating system functions. With horizontal microprogram speed improvements of five to ten times have been observed, sufficient to reduce the time for software implemented operations to a quite acceptable level.

The arguments against microprogramming significant parts of the operating system are:

(1) cost and difficultly of microprogramming,

(2) risk of error and cost of field charges,

(3) differing operating system needs for different projects,

(4) uncertainty as to what will be needed and thus risk of future upgrades,

(5) cost of microprogram storage,

(6) division of organizations into hardware and software teams.

Of these (5) is now negligible, and (1) is not significant if the operating system is to be widely used. Those aware of the real costs of operating system changes will know that the costs of hardware upgrades not significantly greater than for software upgrades. Many of the differences referred to in (3) are due to attempts to avoid or alleviate the performance penalties of a standard operating system, performance penalties that would not be incurred with a microprogrammed operating system. Further, differing operating systems are as great a bar to compatibility and portability as nonstandard hardware. Given the costs in software development that might be placed at risk by an upgrade in fundamental operating system characteristics, it is worth thinking about the design carefully in advance and getting it right, whether in hardware or software. Item (6) need be no obstacle in the right context.

A computer with large parts of its operating System in horizontal microprogram (the GEC4080) was delivered to customers in the U.K. in October 1972. Development was not found to be particularly difficult, the anticipated performance advantages were realized, and no subsequent

problems of errors or inflexibility were encountered. This computer has remained in production since then and is quite popular in the U.K.

We would recommend that a production implementation of TACEXEC should use microprogram to reduce the time required for critical operating system functions.

VIII TOWARDS A HIGH-LEVEL LANGUAGE FOR THE IMPLEMENTATION OF TACEXEC

The design of TACEXEC is based on the concept of abstract data types, defined by the modules of the specifications. Examples of the types defined are:

process,
segment,
category set,
access level,
event.

Ideally, the language used to implement TACEXEC should provide mechanisms to allow the definition of new types, and subsequently the use of such types with the same flexibility as if they had been built in. Unfortunately, very few languages currently provide for the definition of new types by the user, and those few languages are quite unsuitable for TACEXEC implementation. Shortly the Ada Language (DOD-1) will become available, and will provide the needed mechanisms. Until then, it will be necessary to live with existing languages that are less than ideal.

Three other language capabilities are necessary for the implementation of TACEXEC. These are:

access to low level machine facilities,
parallel processing facilities,
exception handling facilities.

The access to low level machine facilities is necessary to allow the TACEXEC implementation to use hardware facilities for protection, relocation, user processes, interrupt handling, etc. Access to low level machine facilities are also necessary for implementing the input-output functions of TACEXEC. These low level facilities are required only in certain very localized sections of program, and thus the

language facility providing this access can be quite crude. It will also be necessary to preclude use of these low level operations by user programs, but this restriction will be imposed by TACEXEC and the hardware, rather than by the language.

TACEXEC needs within itself a number of parallel processes that operate asynchronously. The facility needed is however very basic, for the more elegant and easy to use facilities provided to user programs are constructed by TACEXEC itself. All that is required in order to implement TACEXEC is the ability to describe parallel processes (programs) that share data only explicitly and the ability to construct a binary semaphore so as to provide a program to handle a interrupt. Access to the register and status information of suspended programs, and use of this information to resume such programs--facilities necessary for the dispatching functions of TACEXEC--would presumably be provided by access to low level machine facilities.

TACEXEC makes extensive use of exception returns to indicate invalid use of its facilities, as do all other operating systems. The specifications are written on the basis of an exception handling mechanism distinct from any other results returned by the operations. Such distinction is very helpful to the programmer, to make programs simpler and easier to understand, and also to make the compiled code smaller. However, this distinction between exception returns and normal returns with results is not essential to the functioning of the system.

While a language such as Ada would be ideally suited to the implementation of TACEXEC, it is clear that the final definitive language, and efficient compilers for that language, will not be available for, perhaps, two to three years. Other comparable languages are either only partially developed and do not yet have compilers either, such as Euclid, Modula, Alphard, and CLU, or else require enormous run time support systems much larger than TACEXEC, such as Simula and Algol 68. Consequently, the initial versions of TACEXEC must be implemented in some other existing language.

There is no reason to doubt that other existing languages can be used to implement TACEXEC. Every function required for TACEXEC can be provided, but the programs will be less obvious and less easy to read than in a language specifically intended for building systems from user data types.

For instance, in a language without user defined data types and without type checking, the user defined data types become simply data items or data structures such as arrays, while the operations on the data types become procedures. Of course, the existing compiler will not be able to perform any type checking, thus requiring a specially designed preprocess or much greater care by the programmers to ensure that these operations are only applied to data items of the correct "type".

Access to low level machine facilities, and the operations on semaphores, might preferably also be provided as procedures, the bodies of which would be programmed in assembly language. The use of procedures will cause an additional overhead, but will restrict the assembly code to specific procedures and will reduce its adverse impact on the readability of programs.

Advantage can be taken of the named COMMON, COMPOOL, or equivalent facilities of many languages. The structure of TACEXEC is such that most data structures need be accessed only by a single program module. If this data is declared in a named COMMON in that program module only, then the risk of inadvertent access by other program module is greatly reduced. Unnamed or blank COMMON should never be used.

The mechanisms used to achieve parallel processes in TACEXEC must depend on the details of the particular language implementation. Particular care must be taken over the implementation of local variables and temporary variables, and also over the use of any language run time support procedures. There is a risk that the same data storage locations may be inadvertently used by several processes.

In the absence of an exception handling mechanism, the exception returns from operations must be represented by additional parameters in the results returned. The values of such parameters must be tested before the other results can be used.

In conclusion, recent research has led to an understanding of the features required of a high-level programming language for the efficient realization of system structured as a hierarchy of modules. The most promising language is Ada, but until its appearance several other languages could be used provided the programmer follows certain easily established conventions.

IX CONCLUSIONS AND POSSIBLE FUTURE TASKS

The main products of this investigation are

(1) A design for the kernel of a real time operating system (called TACEXEC) expressed as specifications for the seven modules that comprise the system. The specifications are written in the language SPECIAL.

(2) A mathematical model (developed in part on other SRI contracts) that defines acceptable information transfers among users according to their security level.

(3) A method for proving the specification of TACEXEC with respect to the security model, and the illustration of this method for several of the specifications.

(4) Several algorithms for allocating and scheduling iterative tasks in a multiprogramming environment such that all tasks are guaranteed to obtain service as needed. An informal proof is given that one of these algorithms achieves maximum usage of the system, excluding the overhead time for the scheduler. The algorithm that achieves less than optimum usage at the benefit of a simple scheduling discipline (based on task priority).

We believe that TACEXEC can realize the goals established in Chapter I: (1) capability for handling real-time tasks, (2) adequate functional capability for supporting a variety of subsystems, (3) efficiently implementable, (4) secure, (5) provable and (6) portable. For example, to demonstrate (2) we described an approach to realizing a secure message system using the primitives of TACEXEC.

We believe that TACEXEC in its present form can serve as a practical kernel for many future array real time operating systems. However, since there are problems related to the proof of TACEXEC and to its use in complex configuration, we recommend the following tasks be considered in an extension of the current investigation.

* Multiprocessor Configuration: The design of upper levels of the current TACEXEC is not affected by the number of

hardware processors, which impacts only the level of resources available. Each type of task indigenous to the tactical environment is handled easily by a minicomputer. A multicomputer configuration could be effectively used for situations where the computing requirements of the tasks exhaust the capacity of a single machine. Our present plan is to consider the implementation on a single processor. We propose to generalize the lower levels of the design so that they can execute on a variable number of processors, where suitable multiprocessor hardware is available. The main problems to be considered relate to the management of processing and storage resouces without sacrificing the security or guaranteed performance.

- Network Configuration: All components of the multiprocessor configuration discussed above are assumed to be contiguous. A more general situation would involve a geographical separation of the computers as a network. Many of the problems and solutions associated with multiprocessor systems apply here, except that the low inter-connection bandwidth must be considered in allocating resources to processes. In addition, the security issues are compounded by computer separation, for example:

  (1) crypto graphic techniques might be needed to secure the transmission,

  (2) some computers might be insecure, and hence cannot be fully trusted,

  (3) a computer might fall into enemy hands and thus act in a malicious manner.

- Fault Tolerance: For critical applications it is essential that useful computation continues, even in the presence of hardware faults. Many techniques have been suggested for providing such fault tolerance, particularly for the type of tasks that are our concern. For example, the SIFT concept,[16] enables critical tasks to be processed by two or more processors, provides for a comparison of the result computed by the replicated processes and performs rapid reconfiguration of the system on the detection and location of a fault. We would consider incorporating the SIFT concept into a multiprocessor configuration. One potential disadvantage of the SIFT concept is that it is extravagant in its use of redundancy. This may not be a serious criticism as the cost of the computer hardware diminishes. It may also be possible to alleviate this problem by using redundancy techniques that are more cost effective in special situations, e.g., error correcting coding for storage.

- Recovery From Application Program Errors: We envisage that, by formally verifying the TACEXEC and by incorporating

hardware redundancy into the system, the TACEXEC will be invulnerable to "system" failures. However, some of the application programs may not have been verified, and an error in on application program could have serious effects for that application (though it would not affect any other independent applications). A technique developed at the University of Newcastle, by Brian Randall and Michael Melliar-Smith, addresses this issue. Briefly an acceptance test is provided with the application program, which if not satisfied by a particular invocation causes an alternate version of the program to be invoked. We would include the mechanisms for such detection and recovery within the TACEXEC, and investigate techniques for writing acceptance tests and alternate programs for tactical programs. The original recovery concept would have to be extended to handle asynchronously communicating programs.

* Proof Techniques: The need to produce a complete, implementable TACEXEC design within the current contract precluded the allocation of significant effort to verification of the system. We designed the system so that it is provable, but the actual development of the implementation proofs will require significant effort. In particular, there is a need to prove properties relating to the system's ability to meet the time constraints required of the applicaton programs, a problem that was only partly addressed during this current investigation. Also, the current work is concerned with scheduling for the worst case. Future work should consider a distribution for processing times.

* Extension of SPECIAL: During the investigation the SPECIAL specification language evolved, primarily to express the behavior of a module that is accessed by asynchronous processors. It is not clear that the specification constructs we proposed are formalizable for proof or are adequate for expressing general inter-process communications. Additional effort is needed here that should be preceded by an investigation of a variety of system applications, including distributed systems.

Appendix A

SPECIFICATIONS FOR TACEXEC

Appendix A

SPECIFICATIONS FOR TACEXEC

CONTENTS

```
(INTERFACE TACEXEC
    process_coordination
    user_io
    process_primitives
    file_system
    (virtual_memory WITHOUT create_segment delete_segment
                            get_segment)
)
(INTERFACE TACEXEC_D
    file_system
    virtual_memory
    system_io
    dispatcher
)

(INTERFACE TACEXEC_C
    virtual_memory
    system_io
    dispatcher
)

(INTERFACE TACEXEC_B
    system_io
    dispatcher
)
(INTERFACE TACEXEC_A
    dispatcher
)
```

PROCESS COORDINATION

```
MODULE process_coordination


    TYPES

clearance: { INTEGER i | 0 < i AND i <= max_clearance };
category_set:
{ VECTOR_OF BOOLEAN cs | LENGTH(cs) = number_of_categories };
access_level:
STRUCT_OF(clearance security_clearance;
          category_set security_categories;
          clearance integrity_clearance;
          category_set integrity_categories);
segment_number: {INTEGER sn | 0 <= sn AND sn < segments_per_as};
offset: {INTEGER i | 0 <= i AND i < max_seg_size};


    DEFINITIONS

BOOLEAN read_allowed(access_level subject_al, object_al)
   IS   subject_al.security_clearance
       >= object_al.security_clearance
      AND subject_al.integrity_clearance
         <= object_al.integrity_clearance
     AND(FORALL INTEGER i | 0 < i AND i <= number_of_categories:
            ( object_al.security_categories[i]
             => subject_al.security_categories[i])
           AND( subject_al.integrity_categories[i]
              => object_al.integrity_categories[i]));
BOOLEAN write_allowed(access_level subject_al, object_al)
   IS read_allowed(object_al, subject_al);
access_level seg_access_level(segment s)
   IS SOME access_level l | EXISTS address_space as:
                              EXISTS segment_number sn:
                                   h_as_entry(as, sn, l) = s
```

```
                    AND h_as_entry_owned(as, sn, l)
                       = TRUE
                    AND ~(EXISTS access_level l1 |
                           read_allowed(l, l1):
                               h_as_entry(as, sn, l1)
                              = s
                             AND h_as_entry_owned(as,
                                                  sn,
                                                  l1)
                               = TRUE);
```

```
    EXTERNALREFS


    FROM virtual_memory:
address_space, segment: DESIGNATOR;
INTEGER max_clearance $( the highest clearance) ,
       number_of_categories,
       segments_per_as $(tne number of possible
                         segments in an address space),
       max_seg_size $(the maximum size of a segment);
VFUN h_as_exists(address_space as; access_level l) -> BOOLEAN b;
VFUN h_as_entry(address_space as; segment_number sn; access_level l)
               -> segment s;
VFUN h_as_entry_owned(address_space as; segment_number sn;
                      access_level l) -> BOOLEAN b;
VFUN h_read(segment s; offset os; access_level l)
          -> INTEGER contents;
OFUN segment_write(segment_number sn; offset os;
                   INTEGER contents)
                  [address_space as; access_level al];
```

```
    FUNCTIONS

OFUN P(segment_number sn; offset os)
      [address_space as; access_level al];
     $( Return if value of s was greater than 0 with value
        of s decremented by 1)
    DEFINITIONS
       segment s IS h_as_entry(as, sn, al);
    EXCEPTIONS
       h_as_entry_owned(as, sn, al) = ?;
       s = ? OR seg_access_level(s) ~= al;
       h_read(s, os, seg_access_level(s)) = ?;
    DELAY UNTIL h_read(s, os, seg_access_level(s)) > 0;
    ASSERTIONS
       h_as_exists(as, al);
    EFFECTS
       EFFECTS_OF
        segment_write(sn, os,
                      h_read(s, os, seg_access_level(s))-1,
                      as, al);

OFUN V(segment_number sn; offset os)
      [address_space as;access_level al];
     $( Increment the value of semaphore s)
    DEFINITIONS
       segment s IS h_as_entry(as, sn, al);
    EXCEPTIONS
       h_as_entry_owned(as, sn, al) = ?;
       s ~= ? AND read_allowed(al, seg_access_level(s))
        AND ~write_allowed(al, seg_access_level(s));
       s ~= ? AND read_allowed(al, seg_access_level(s))
        AND h_read(s, os, seg_access_level(s)) = ?;
    ASSERTIONS
       h_as_exists(as, al);
    EFFECTS
```

```
        s ~= ? AND write_allowed(al, seg_access_level(s))
        AND h_read(s, os, seg_access_level(s)) ~= ?
       => EFFECTS_OF
           segment_write(sn, os,
                         h_read(s, os, seg_access_level(s))+1,
                         as, al);

END_MODULE
```

```
MODULE user_io


    TYPES

clearance: { INTEGER i | 0 < i AND i <= max_clearance };
category_set:
{ VECTOR_OF BOOLEAN cs | LENGTH(cs) = number_of_categories };
access_level:
STRUCT_OF(clearance security_clearance;
          category_set security_categories;
          clearance integrity_clearance;
          category_set integrity_categories);


    PARAMETERS

access_level h_device_al(INTEGER dev_ind)  $( access level of each
                                              device);


    DEFINITIONS

BOOLEAN read_allowed(access_level subject_al, object_al)
   IS   subject_al.security_clearance
        >= object_al.security_clearance
      AND subject_al.integrity_clearance
          <= object_al.integrity_clearance
     AND(FORALL INTEGER i | 0 < i AND i <= number_of_categories:
            ( object_al.security_categories[i]
              => subject_al.security_categories[i])
           AND( subject_al.integrity_categories[i]
              => object_al.integrity_categories[i]));
BOOLEAN write_allowed(access_level subject_al, object_al)
   IS read_allowed(object_al, subject_al);
```

```
    EXTERNALREFS


    FROM process_primitives:
event: DESIGNATOR;
OFUN occurrence(event e)
              [access_level al] $( Event e has occurred) ;

    FROM virtual_memory:
INTEGER max_clearance $( the highest clearance) ,
       number_of_categories;


    FUNCTIONS


VFUN h_device_event(INTEGER dev_ind; access_level al) -> event e;
          $( Returns the event which can occur when device status
           changes)
    HIDDEN;
    INITIALLY
       e = ?;

VFUN h_input(INTEGER dev_ind; access_level al) -> INTEGER data;
          $( The current input from the device)
    HIDDEN;
    INITIALLY
       data = ?;

VFUN h_output(INTEGER dev_ind; access_level al) -> INTEGER data;
          $( The next output to the device)
    HIDDEN;
    INITIALLY
       data = ?;
```

```
VFUN h_command(INTEGER dev_ind; access_level al)
              -> INTEGER command;  $( The next command for the
                                      device)
    HIDDEN;
    INITIALLY
       command = ?;

VFUN h_status(INTEGER dev_ind; access_level al) -> INTEGER status;
          $( The current status of the device)
    HIDDEN;
    INITIALLY
       status = ?;

OFUN set_event(INTEGER dev_ind; event e)[access_level al];
          $( When status changes event e may occur)
    EXCEPTIONS
       ~ write_allowed(al, n_device_al(dev_ind));
    EFFECTS
       'n_device_event(dev_ind, h_device_al(dev_ind)) = e;

OVFUN read_device(INTEGER dev_ind)[access_level al]
                -> INTEGER data; $( Read data from device)
    EXCEPTIONS
       al ~= n_device_al(dev_ind);
       n_input(dev_ind, al) = ?;
    EFFECTS
       data = h_input(dev_ind, al);
       'n_input(dev_ind, al) = ?;

OFUN write_device(INTEGER dev_ind; INTEGER data)[access_level al];
          $( Output data to device)
    EXCEPTIONS
       al ~= h_device_al(dev_ind);
       h_output(dev_ind, al) ~= ?;
    EFFECTS
```

```
        'h_output(dev_ind, al) = data;

OFUN send_command(INTEGER dev_ind; INTEGER command)
                [access_level al]; $( Give command to device)
    EXCEPTIONS
       al ~= h_device_al(dev_ind);
       h_command(dev_ind, al) ~= ?;
    EFFECTS
       'h_command(dev_ind, al) = command;

VFUN receive_status(INTEGER dev_ind)[access_level al]
                -> INTEGER status; $( Get the devices status)
    EXCEPTIONS
       ~ read_allowed(al, h_device_al(dev_ind));
       h_status(dev_ind, al) = ?;
    DERIVATION
       h_status(dev_ind, al);

OVFUN device_output()[INTEGER dev_ind] -> INTEGER data;
          $( Device reads data it is to output)
    EXCEPTIONS
       h_output(dev_ind, h_device_al(dev_ind)) = ?;
    EFFECTS
       data = h_output(dev_ind, h_device_al(dev_ind));
       'h_output(dev_ind, h_device_al(dev_ind)) = ?;

OFUN device_input(INTEGER data)[INTEGER dev_ind];
          $( Device places input data into input buffer)
    EXCEPTIONS
       h_input(dev_ind, h_device_al(dev_ind)) ~= ?;
    EFFECTS
       'h_input(dev_ind, h_device_al(dev_ind)) = data;
```

```
OVFUN device_command()[INTEGER dev_ind] -> INTEGER command;
          $( The device gets a command)
    EXCEPTIONS
       h_command(dev_ind, h_device_al(dev_ind)) = ?;
    EFFECTS
       command = h_command(dev_ind, h_device_al(dev_ind));
       'h_command(dev_ind, h_device_al(dev_ind)) = ?;

OFUN change_status(INTEGER status; BOOLEAN occur)
                  [INTEGER dev_ind];  $( Reset the status of the
                                         device)
    EFFECTS
       FORALL access_level l | read_allowed(l,
                                            h_device_al(dev_ind)):
          'h_status(dev_ind, l) = status;
        ( occur
         AND h_device_event(dev_ind, h_device_al(dev_ind)) ~= ?)
       => EFFECTS_OF occurrence(h_device_event(dev_ind,
                                               h_device_al(dev_ind)
),
                                h_device_al(dev_ind));

END_MODULE
```

```
MODULE process_primitives


    TYPES

process, event: DESIGNATOR;
machine_state: DESIGNATOR;
process_type: { iterative, demand, background };
segment_number: { INTEGER sn | 0 <= sn AND sn < segments_per_as };
offset: { INTEGER i | 0 <= i AND i < max_seg_size };
program_counter: STRUCT_OF(segment_number sn; offset of);
process_info:
STRUCT_OF(process_type type;
          INTEGER next_service;
          INTEGER interval;
          INTEGER duration;
          INTEGER deadline;
          INTEGER processing_remaining;
          event ev;
          BOOLEAN running;
          address_space as;
          program_counter pc;
          machine_state ms);
clearance: { INTEGER i | 0 < i AND i <= max_clearance };
category_set:
{ VECTOR_OF BOOLEAN cs | LENGTH(cs) = number_of_categories };
access_level:
STRUCT_OF(clearance security_clearance;
          category_set security_categories;
          clearance integrity_clearance;
          category_set integrity_categories);
```

PARAMETERS

```
INTEGER start_time $( time when system is initialized) ;
program_counter initial_pc  $( address of first instruction in
                               first process);
machine_state initial_ms $( the initial state of logical machine
                            registers for each new process);
```

DEFINITIONS

```
BOOLEAN time_critical_process(process p; access_level l)
   IS h_process_info(p, l).type = iterative
      OR h_process_info(p, l).type = demand;

access_level initial_level
   IS STRUCT(1,
             VECTOR(FOR i FROM 1 TO number_of_categories: FALSE),
             max_clearance,
             VECTOR(FOR i FROM 1 TO number_of_categories: TRUE));
BOOLEAN read_allowed(access_level subject_al, object_al)
   IS   subject_al.security_clearance
       >= object_al.security_clearance
      AND subject_al.integrity_clearance
         <= object_al.integrity_clearance
     AND(FORALL INTEGER i | 0 < i AND i <= number_of_categories:
             ( object_al.security_categories[i]
              => subject_al.security_categories[i])
            AND( subject_al.integrity_categories[i]
               => object_al.integrity_categories[i]));
BOOLEAN write_allowed(access_level subject_al, object_al)
   IS read_allowed(object_al, subject_al);
```

```
    EXTERNALREFS

    FROM virtual_memory:
address_space: DESIGNATOR;
INTEGER max_clearance, number_of_categories, segments_per_as,
       max_seg_size;
VFUN h_as_exists(address_space as; access_level al) -> BOOLEAN b;
OFUN get_segment(address_space source_as;
                 access_level source_al;
                 segment_number source_sn;
                 segment_number dest_sn)
                [address_space as; access_level al];


    ASSERTIONS

FORALL process p; access_level l | time_critical_process(p, l):
  h_process_info(p, l).processing_remaining >= 0;


    FUNCTIONS


VFUN h_process_exists(process p; access_level al) -> BOOLEAN b;
          $( True if process p exists)
    HIDDEN;
    INITIALLY
         CARDINALITY({ process p | h_process_exists(p,
                                                    initial_level)
})
        = 1
       AND(FORALL access_level l | l ~= initial_level:
             FORALL process p: h_process_exists(p, l) ~= TRUE);
```

```
VFUN h_process_info(process p; access_level al) -> process_info pi;
        $( Returns scheduling information about a given process)
    HIDDEN;
    INITIALLY
        pi
       =(IF h_process_exists(p, al)
           THEN STRUCT(background, ?, ?, ?, ?, ?, ?, ?,
                       (SOME address_space as |
                         h_as_exists(as, initial_level)),
                       initial_pc, initial_ms)
           ELSE ?);

VFUN h_event_min_period(event e; access_level al)
                       -> INTEGER min_period;
        $( True if the event e exists)
    HIDDEN;
    INITIALLY
       FORALL event e:
        CARDINALITY({access_level l | h_event_min_period(e, l) = -1})
         = 1
        AND (FORALL access_level l | h_event_min_period(e, l) ~= -1:
             h_event_min_period(e, l) = ?);

VFUN h_time() -> INTEGER time; $( Clock time)
    HIDDEN;
    INITIALLY
       time = start_time;

OVFUN create_event(INTEGER min_period)[access_level al] -> event e;
      $( Create a new event type)
    EXCEPTIONS
       min_period <= 0;
    EFFECTS
       h_event_min_period(e, al) = -1;
       'h_event_min_period(e, al) = min_period;
```

```
OPUN block()[process p; access_level al];
     $( The process running wishes to relinquish the processor)
    EXCEPTIONS
       ~time_critical_process(p, al);
    DELAY WITH 'h_process_info(p, al).running = FALSE;
          UNTIL h_process_info(p, al).next_service < h_time()
                AND h_time()
                    <= h_process_info(p, al).next_service
                       + h_process_info(p, al).interval
                       - h_process_info(p, al).duration;
    EFFECTS
        h_process_info(p, al).type = iterative
       =>  'h_process_info(p, al).next_service > h_time()
          AND  'h_process_info(p, al).next_service
              - h_process_info(p, al).interval
             < h_time()
         AND(EXISTS INTEGER n:
                  h_process_info(p, al).next_service
                + n * h_process_info(p, al).interval
                = 'h_process_info(p, al).next_service);
        h_process_info(p, al).type = demand
       => 'h_process_info(p, al).next_service = ?;
        'h_process_info(p, al).processing_remaining
       = h_process_info(p, al).duration;
        'h_process_info(p, al).deadline
       = h_process_info(p, al).next_service
         + h_process_info(p, al).interval;
       'h_process_info(p, al).running = TRUE;

OPUN tick()[access_level al];  $( Time passes so increment the
                                  clock)
    ASSERTIONS
       al = initial_level;
    EFFECTS
       'h_time() = h_time() + 1;
```

```
      FORALL process p;access_level l |
        time_critical_process(p, al)
        AND h_process_info(p, al).running:
         IF h_process_info(p, al).processing_remaining
            < h_process_info(p, al).deadline - h_time()
          THEN 'h_process_info(p, al).processing_remaining
               = h_process_info(p, al).processing_remaining
               OR 'h_process_info(p, al).processing_remaining
                  = h_process_info(p, al).processing_remaining - 1
          ELSE 'h_process_info(p, al).processing_remaining
               = h_process_info(p, al).processing_remaining;

OVFUN create_iterative_process(INTEGER int;
                               INTEGER dur;
                               address_space nas;
                               program_counter npc;
                               access_level l)
                              [access_level al]
                              -> process p;
        $( Permit the process p to be scheduled and run as an
         iterative process)
    EXCEPTIONS
       dur < 0;
       int < dur;
       ~ write_allowed(al, l);
       read_allowed(al, l) AND ~h_as_exists(nas, l);
       RESOURCE_ERROR;
    EFFECTS
        h_as_exists(nas, l)
       =>  h_process_exists(p, l) = FALSE
          AND 'h_process_exists(p, l) = TRUE
         AND 'h_process_info(p, l)
            = STRUCT(iterative,
                     (SOME INTEGER i | h_time() <= i
                                       AND i < h_time() + int),
```

```
                    int, dur, ?, 0, ?, FALSE, nas, npc, initial_ms);

OVFUN create_demand_process(INTEGER dur;
                            event e;
                            address_space nas;
                            program_counter npc;
                            access_level l)
                           [access_level al]
                           -> process p;  $( Permit the process p
                                             to run whenever event e
                                             occurs)
    DEFINITIONS
       access_level el
        IS SOME access_level el | read_allowed(al, el)
                                  AND h_event_min_period(e, el) > 0;
    EXCEPTIONS
       el = ?;
       h_event_min_period(e, el) <= dur;
       dur < 0;
       ~ write_allowed(al, l);
       read_allowed(al, l) AND ~h_as_exists(nas, l);
       RESOURCE_ERROR;
    EFFECTS
        h_as_exists(nas, l)
       =>  h_process_exists(p, l) = FALSE
          AND 'h_process_exists(p, l) = TRUE
         AND 'h_process_info(p, l)
            = STRUCT(demand, ?, h_event_min_period(e, el),
                     dur, ?, 0, e, FALSE, nas, npc, initial_ms);

OVFUN create_background_process(address_space nas;
                                program_counter npc;
                                access_level l)
                               [access_level al]
                               -> process p;
```

```
    EXCEPTIONS
       ~ write_allowed(al, l);
       read_allowed(al, l) AND ~h_as_exists(nas, l);
       RESOURCE_ERROR;
    EFFECTS
        h_as_exists(nas, l)
       =>  h_process_exists(p, l) = FALSE
          AND 'h_process_exists(p, l) = TRUE
         AND 'h_process_info(p, l)
            = STRUCT(background, ?, ?, ?, ?, ?, ?, ?,
                     nas, npc, initial_ms);

OFUN delete_process(process p; access_level l)[access_level al];
          $( The process p at level l no longer should exists)
    EXCEPTIONS
       ~ write_allowed(al, l);
       ~(read_allowed(al, l) => h_process_info(p, l) ~= ?);
    EFFECTS
       'h_process_info(p, l) = ?;

OFUN occurrence(event e)[access_level al];  $( Wake up processes
                                               waiting for event e)
    EXCEPTIONS
       ~(EXISTS access_level el | read_allowed(al, el):
            h_event_min_period(e, el) > 0);
    ASSERTIONS
       FORALL access_level l | write_allowed(al, l):
          FORALL process p | h_process_info(p, l).ev = e:
             h_process_info(p, l).deadline <= h_time();
    EFFECTS
       FORALL access_level l | write_allowed(al, l):
          FORALL process p | h_process_info(p, l).ev = e:
             'h_process_info(p, l).next_service = h_time();
```

```
OPUN segment_get(process source_p;
                 segment_number source_sn;
                 access_level source_al;
                 segment_number dest_sn)
                [process p; access_level al];
    EXCEPTIONS
       read_allowed(al, source_al)
        AND h_process_info(source_p, source_al) = ?;
       EXCEPTIONS_OF
        get_segment(h_process_info(source_p, source_al).as,
                    source_al, source_sn, dest_sn,
                    h_process_info(p, al).as,
                    al);
    ASSERTIONS
       h_process_info(p, al) ~= ?;
    EFFECTS
       EFFECTS_OF
        get_segment(h_process_info(source_p, source_al). as,
                    source_al, source_sn, dest_sn,
                    h_process_info(p, al).as,
                    al);

END_MODULE
```

```
MODULE file_system


    TYPES

clearance: { INTEGER i | 0 < i AND i <= max_clearance };
category_set:
{ VECTOR_OF BOOLEAN cs | LENGTH(cs) = number_of_categories };
access_level:
STRUCT_OF(clearance security_clearance;
          category_set security_categories;
          clearance integrity_clearance;
          category_set integrity_categories);
segment_number: { INTEGER sn | 0 <= sn AND sn < segments_per_as };
offset: { INTEGER i | 0 <= i AND i < max_seg_size };
segment_bound: { INTEGER i | 0 <= i AND i <= max_seg_size };
name: {VECTOR_OF CHAR n | LENGTH(n) <= name_length};
file_address: STRUCT_OF(name nm; INTEGER off);
word: ONE_OF (INTEGER, name, process,
              event, wakeup, processor, address_space);


    PARAMETERS

INTEGER max_files(access_level l) $( the most files at level l) ,
       max_level_size(access_level l)  $( the amount of storage at
                                          level l);


    DEFINITIONS

BOOLEAN read_allowed(access_level subject_al, object_al)
   IS   subject_al.security_clearance
      >= object_al.security_clearance
     AND subject_al.integrity_clearance
```

```
        <= object_al.integrity_clearance
     AND(FORALL INTEGER i | 0 < i AND i <= number_of_categories:
            ( object_al.security_categories[i]
             => subject_al.security_categories[i])
           AND( subject_al.integrity_categories[i]
              => object_al.integrity_categories[i]));
BOOLEAN write_allowed(access_level subject_al, object_al)
   IS read_allowed(object_al, subject_al);
access_level seg_access_level(segment s)
   IS SOME access_level l | EXISTS address_space as:
                              EXISTS segment_number sn:
                                   h_as_entry(as, sn, l) = s
                                  AND h_as_entry_owned(as, sn, l)
                                     = TRUE
                                 AND ~(EXISTS access_level l1 |
                                        read_allowed(l, l1):
                                            h_as_entry(as, sn, l1)
                                           = s
                                          AND h_as_entry_owned(as,
                                                               sn,
                                                               l1)
                                             = TRUE);
INTEGER segment_size(segment s)
   IS CARDINALITY({ offset i | h_read(s, i, seg_access_level(s))
                              ~= ? });
SET_OF access_level read_name_set(name n; access_level rl, al)
   IS { access_level l |   read_allowed(rl, l)
                         AND read_allowed(al, l)
                        AND h_file_contents(n, l) ~= ?
                       AND ~(EXISTS access_level l1 |
read_allowed(rl, l1) AND read_allowed(l1, l):
                                h_file_contents(n, l1) ~= ?) };
SET_OF access_level write_name_set(name n; access_level wl, al)
   IS { access_level l |   write_allowed(wl, l)
                         AND write_allowed(al, l)
```

```
                        AND h_file_contents(n, l) ~= ?
                       AND ~(EXISTS access_level l1 |
write_allowed(wl, l1) AND write_allowed(l1, l):
                                  h_file_contents(n, l1) ~=  ;;
INTEGER level_size(access_level l)
   IS CARDINALITY({ file_address addr | h_file_contents(addr.nm, l)
[addr.off]
                                       ~= ? });


    EXTERNALREFS


    FROM virtual_memory:
address_space, segment: DESIGNATOR;
INTEGER max_seg_size $( the maximum size of a segment) ,
       max_clearance $( the highest clearance) ,
       number_of_categories,
       segments_per_as  $( the maximum number of segments in an
                          address space),
       name_length $(number of characters in a name);
VFUN h_as_exists(address_space as; access_level al) -> BOOLEAN b;
VFUN h_read(segment s; offset i; access_level l) -> word c;
VFUN h_as_entry(address_space as;
                segment_number sn;
                access_level l)
               -> segment s;
VFUN h_as_entry_owned(address_space as; segment_number sn;
                      access_level l)
                     -> BOOLEAN b;
OFUN segment_create(segment_number sn;
                    segment_bound i;
                    VECTOR_OF word initial_contents)
                   [address_space as; access_level al];
```

```
    FROM dispatcher:
event, process, wakeup, processor: DESIGNATOR;


    FUNCTIONS


VFUN h_file_contents(name n; access_level l)
                    -> VECTOR_OF word contents;  $( The contents
                                                    of a file)
    HIDDEN;
    INITIALLY
       contents = ?;

OFUN create_file(name n)[access_level al]; $( Create a new file)
    EXCEPTIONS
       h_file_contents(n, al) ~= ?;
        CARDINALITY({ name n | h_file_contents(n, al) ~= ? })
       >= max_files(al);
    EFFECTS
       'h_file_contents(n, al) = VECTOR();

OFUN delete_file(name n)[access_level al];  $( Delete an existing
                                               file)
    EXCEPTIONS
       h_file_contents(n, al) = ?;
    EFFECTS
       'h_file_contents(n, al) = ?;

OFUN load_file(name n; access_level rl; segment_number sn)
              [address_space as; access_level al];
          $( Create a new segment with segment number sn with the
           contents of file with name r)
    DEFINITIONS
       access_level l
```

```
        IS IF CARDINALITY(read_name_set(n, rl, al)) = 1
            THEN SOME access_level l INSET read_name_set(n, rl,
                                                         al)
            ELSE ?;
    EXCEPTIONS
       l = ?;
       EXCEPTIONS_OF segment_create(sn,
                                    LENGTH(h_file_contents(n, l)),
                                    VECTOR(), as, al);
    ASSERTIONS
       h_as_exists(as, al);
    EFFECTS
       EFFECTS_OF segment_create(sn,
                                 LENGTH(h_file_contents(n, l)),
                                 h_file_contents(n, l), as, al);

OFUN unload_file(name n; access_level wl; segment_number sn)
                 [address_space as; access_level al];
        $( Copy the contents of segment sn into file with name n)
    DEFINITIONS
       segment s IS h_as_entry(as, sn, al);
       access_level l
          IS IF CARDINALITY(write_name_set(n, wl, al)) = 1
              THEN SOME access_level l INSET write_name_set(n, wl,
                                                            al)
              ELSE ?;
    EXCEPTIONS
       h_as_entry_owned(as, sn, al) = ?;
       s = ? OR ~read_allowed(al, seg_access_level(s));
       ~( l = ?
         =>( read_allowed(al, l)
            =>  segment_size(s) + level_size(l)
              - LENGTH(h_file_contents(n, l))
             <= max_level_size(l)));
    ASSERTIONS
```

```
      h_as_exists(as, al);
    EFFECTS
        l ~= ?
       AND  segment_size(s) + level_size(l)
          - LENGTH(h_file_contents(n, l))
          > max_level_size(l)
      => 'h_file_contents(n, l)
        = VECTOR(FOR i FROM 0 TO segment_size(s) - 1
                 : h_read(s, i, al));

OFUN append_to_file(name n; access_level wl; segment_number sn)
                    [address_space as; access_level al];
    $(append the contents of a segment to a file with name n)
    DEFINITIONS
      segment s IS h_as_entry(as, sn, al);
      access_level l
         IS IF CARDINALITY(write_name_set(n, wl, al)) = 1
            THEN SOME access_level l INSET write_name_set(n, wl,
                                                          al)

            ELSE ?;
    EXCEPTIONS
      h_as_entry_owned(as, sn, al) = ?;
      s = ? OR ~read_allowed(al, seg_access_level(s));
      ~( l = ?
        =>( read_allowed(al, l)
           => segment_size(s) + level_size(l)
             <= max_level_size(l)));
    EFFECTS
        l ~= ?
       AND segment_size(s) + level_size(l)
          > max_level_size(l)
      => 'h_file_contents(n, l)
        = VECTOR(FOR i
                 FROM 0
                 TO LENGTH(h_file_contents(n, l))
```

```
        + segment_size(s) - 1:
      IF i < LENGTH(h_file_contents(n, l))
       THEN h_file_contents(n, l)[i]
       ELSE h_read(s,
                   i-LENGTH(h_file_contents(n, l)),
                   al));

END_MODULE
```

```
MODULE virtual_memory


    TYPES

address_space, segment: DESIGNATOR;
clearance: { INTEGER i | 0 < i AND i <= max_clearance };
category_set:
{ VECTOR_OF BOOLEAN cs | LENGTH(cs) = number_of_categories };
access_level:
STRUCT_OF(clearance security_clearance;
          category_set security_categories;
          clearance integrity_clearance;
          category_set integrity_categories);
segment_number: {INTEGER sn | 0 <= sn AND sn < segments_per_as };
segment_bound: { INTEGER i | 0 <= i AND i <= max_seg_size };
offset: { INTEGER i | 0 <= i AND i < max_seg_size };
name: {VECTOR_OF CHAR n | LENGTH(n) <= name_length};
word: ONE_OF(INTEGER, name, event, process,
             wakeup, processor, address_space);


    PARAMETERS

INTEGER max_clearance $( the highest clearance) ,
       number_of_categories,
       segments_per_as  $( the maximum number of segments in an
                        address space),
       max_seg_size,
       max_as(access_level al)  $( the most address spaces
                                permitted at a level),
       max_size(access_level al)  $( the amount of memory that can
                                  be consumed at each level),
       initial_as_size $( the size of the initial existing
                        address space ),
```

```
        name_length $( the number of characters in a name);

VECTOR_OF word initial_segment $( contents of initially
                                        existing segment );


    DEFINITIONS

access_level initial_level
   IS STRUCT(1,
             VECTOR(FOR i FROM 1 TO number_of_categories: FALSE),
             max_clearance,
             VECTOR(FOR i FROM 1 TO number_of_categories: TRUE));
BOOLEAN read_allowed(access_level subject_al, object_al)
   IS   subject_al.security_clearance
       >= object_al.security_clearance
      AND subject_al.integrity_clearance
         <= object_al.integrity_clearance
     AND(FORALL INTEGER i | 0 < i AND i <= number_of_categories:
             ( object_al.security_categories[i]
              => subject_al.security_categories[i])
            AND( subject_al.integrity_categories[i]
                => object_al.integrity_categories[i]));
BOOLEAN write_allowed(access_level subject_al, object_al)
   IS read_allowed(object_al, subject_al);
access_level seg_access_level(segment s)
   IS SOME access_level l
       | EXISTS address_space as; segment_number sn:
            h_as_entry(as, sn, l) = s
          AND h_as_entry_owned(as, sn, l) = TRUE
          AND ~(EXISTS access_level l1 | read_allowed(l, l1):
                  h_as_entry(as, sn, l1) = s
                 AND h_as_entry_owned(as, sn, l1) = TRUE);
INTEGER number_as(access_level al)
   IS CARDINALITY({address_space as | h_as_exists(as, al)});
INTEGER segment_size(segment s)
```

```
   IS CARDINALITY({offset i
                  | h_read(s, i, seg_access_level(s))~= ? });
INTEGER total_as_size(address_space as; access_level asl)
   IS SUM(VECTOR(FOR sn FROM 1 TO segments_per_as - 1
                 : IF h_as_entry_owned(as, sn, asl)
                    THEN segment_size(h_as_entry(as, sn, asl))
                    ELSE 0));
INTEGER total_size(SET_OF address_space sas)
   IS IF sas = {}
       THEN 0
       ELSE LET address_space as | as INSET sas
            IN total_as_size(as) + total_size(sas DIFF {as});


    EXTERNALREFS

    FROM dispatcher:
event, process, wakeup, processor: DESIGNATOR;


    ASSERTIONS

max_as(initial_level) > 0;
segments_per_as > 0;
LENGTH(initial_segment) <= max_seg_size;
initial_as_size <= max_size(initial_level);
LENGTH(initial_segment) <= initial_as_size;


    FUNCTIONS

VFUN h_as_exists(address_space as; access_level al) -> BOOLEAN b;
         $( True if address space as is known to exist from level
          al)
    HIDDEN;
```

```
    INITIALLY
       b = h_as_used(as, al);

VFUN h_as_used(address_space as; access_level al) -> BOOLEAN b;
     $( True if address space as has ever existed)
    HIDDEN;
    INITIALLY
       CARDINALITY({address_space as |
                    h_as_used(as, initial_level)})
        = 1
       AND (FORALL address_space as;
                  access_level al | al ~= initial_level:
            ~h_as_used(as, al));

VFUN h_as_size(address_space as; access_level al) -> INTEGER i;
          $( Number of memory words in address space as)
    HIDDEN;
    INITIALLY
       i = (IF h_as_used(as, al)
            THEN initial_as_size
            ELSE ?);

VFUN h_as_entry(address_space as;
                segment_number sn;
                access_level al)
               -> segment s;  $( Returns the segment with segment
                                 number sn in address space as)
    HIDDEN;
    INITIALLY
       IF h_as_used(as, al) AND sn = 0
        THEN h_seg_used(s)
        ELSE s = ?;
```

```
VFUN h_as_entry_owned(address_space as;
                      segment_number sn;
                      access_level al)
                     -> BOOLEAN b;  $( True if segment owned by
                                       address space as)
    HIDDEN;
    INITIALLY
       b = (IF h_as_used(as, al) AND sn = 0 THEN TRUE ELSE ?);


VFUN h_seg_used(segment s) -> BOOLEAN b;  $( True if segment s has
                                             ever existed)
    HIDDEN;
    INITIALLY
       CARDINALITY({segment s ¦ h_seg_used(s)}) = 1;

VFUN h_read(segment s; offset i; access_level al) -> word c;
         $( Returns contents of a word of a segment s)
    HIDDEN;
    INITIALLY
       c = (IF h_seg_used(s) AND read_allowed(al, initial_level)
            THEN initial_segment[i]
            ELSE ?);

OVFUN create_address_space(INTEGER as_size; access_level new_al)
                          [access_level al]
                          -> address_space new_as;
         $( Creates a new address space at access level new_al
          with size as_size)
    EXCEPTIONS
       ~ write_allowed(al, new_al);
        read_allowed(al, new_al)
       AND number_as(new_al) >= max_as(new_al);
        read_allowed(al, new_al)
       AND total_size({address_space as ¦ h_as_exists(as, new_al)})
```

```
            + as_size > max_size(new_al);
    EFFECTS
       h_as_used(new_as, new_al) = FALSE;
       'h_as_used(new_as, new_al) = TRUE;
        ( number_as(new_al) < max_as(new_al)
         AND total_size({address_space as | h_as_exists(as, new_al)})
              + as_size <= max_size(new_al))
       => 'h_as_exists(new_as, new_al) = TRUE
          AND 'h_as_size(new_as, new_al) = as_size;

OFUN delete_address_space(address_space old_as; access_level asl)
                          [access_level al];
          $( Deletes the address space as at level asl)
    EXCEPTIONS
       ~write_allowed(al, asl);
       read_allowed(al, asl) AND ~h_as_exists(old_as, al);
    EFFECTS
        h_as_exists(old_as, asl)
       => 'h_as_exists(old_as, asl) = FALSE
          AND (FORALL segment_number sn |
                h_as_entry_owned(old_as, sn, asl) ~= ?:
                 EFFECTS_OF segment_delete(sn, old_as, asl));

OFUN create_segment(address_space as;
                    access_level asl;
                    segment_number sn;
                    segment_bound i;
                    VECTOR_OF word initial_contents)
                   [access_level al];  $( Creates a segment with
                                          segment number sn in
                                          address space as)
    EXCEPTIONS
       i < 0;
       ~write_allowed(al, asl);
       read_allowed(al, asl) AND h_as_entry_owned(as, sn, asl) ~= ?;
```

```
        read_allowed(al, asl)
         AND total_as_size(as, asl) + i > h_as_size(as, asl);
    EFFECTS
         h_as_entry_owned(as, sn, asl) = ?
         AND total_as_size(as, asl) + i <= h_as_size(as, asl)
        =>(EXISTS segment s:
            h_seg_used(s) = FALSE AND 'h_seg_used(s) = TRUE
            AND 'h_as_entry_owned(as, sn, asl) = TRUE
            AND (FORALL access_level l | read_allowed(l, asl):
                  'h_as_entry(as, sn, l) = s
                 AND (FORALL offset j | 0 <= j AND j < i:
                       'h_read(s, j, l)
                      = (IF initial_contents[j] = ?
                         THEN 0
                         ELSE initial_contents[j]))));

OFUN delete_segment(address_space as; access_level asl;
                    segment_number sn)
                   [access_level al]; $( Delete segment sn)
    EXCEPTIONS
        ~write_allowed(al, asl);
        read_allowed(al, asl)
         AND h_as_entry_owned(as, sn, al) = ?;
    EFFECTS
         h_as_entry_owned(as, sn, asl) ~= ?
        => 'h_as_entry_owned(as, sn, asl) = ?
          AND
          (h_as_entry_owned(as, sn, asl)
           => (FORALL access_level l | read_allowed(l, asl):
                 'h_as_entry(as, sn, asl) = ?
                AND (FORALL offset i:
                 'h_read(h_as_entry(as, sn, asl), i, l) = ?)));
```

```
VFUN segment_read(segment_number sn; offset i)
                  [address_space as; access_level al]
                  -> word c;  $( Returns contents of word i of
                                 segment sn)
    EXCEPTIONS
       h_as_entry_owned(as, sn, al) = ?;
       h_as_entry(as, sn, al) = ?;
       h_read(h_as_entry(as, sn, al), i, al) = ?;
    ASSERTIONS
       h_as_exists(as, al);
    DERIVATION
       h_read(h_as_entry(as, sn, al), i, al);

OFUN segment_write(segment_number sn; offset i; word c)
                   [address_space as; access_level al];
          $( Write data c into a word i of segment sn)
    DEFINITIONS
       segment s IS SOME segment s1 | (EXISTS access_level l:
                     h_as_entry(as, sn, l) = s1);
    EXCEPTIONS
       h_as_entry_owned(as, sn, al) = ?;
       s ~= ? AND read_allowed(al, seg_access_level(s))
        AND h_read(s, i, seg_access_level(s)) = ?;
       s ~= ? AND read_allowed(al, seg_access_level(s))
        AND ~write_allowed(al, seg_access_level(s));
    ASSERTIONS
       h_as_exists(as, al);
    EFFECTS
        s ~= ? AND h_read(s, i, seg_access_level(s)) ~= ?
        AND write_allowed(al, seg_access_level(s))
       => (FORALL access_level l |
           read_allowed(l, seg_access_level(s)):
            'h_read(s, i, l) = c);
```

```
OPUN get_segment(address_space source_as;
                 access_level asl;
                 segment_number source_sn;
                 segment_number dest_sn)
                [address_space as; access_level al];
        $( Allow the segment in address space source_as with
         number source sn to be accessed in the current address
         space with number dest_sn)
    EXCEPTIONS
       read_allowed(al, asl) AND ~h_as_exists(source_as, asl);
       read_allowed(al, asl)
        AND h_as_entry_owned(source_as, source_sn, asl) = ?;
       h_as_entry(as, dest_sn, al) ~= ?;
    ASSERTIONS
       h_as_exists(as, al);
    EFFECTS
       FORALL access_level l
        | read_allowed(l, asl) AND read_allowed(l, al):
          'h_as_entry(as, dest_sn, l)
         = h_as_entry(source_as, source_sn, asl);
       'h_as_entry_owned(as, dest_sn, al) = FALSE;

OPUN segment_create(segment_number sn;
                    segment_bound i;
                    VECTOR_OF INTEGER initial_contents)
                   [address_space as; access_level al];
    EXCEPTIONS
       EXCEPTIONS_OF create_segment(as, al, sn, i,
                                    initial_contents, al);
    ASSERTIONS
       h_as_exists(as, al);
    EFFECTS
       EFFECTS_OF create_segment(as, al, sn, i,
                                 initial_contents, al);
```

```
OFUN segment_delete(segment_number sn)
                    [address_space as; access_level al];
    EXCEPTIONS
       EXCEPTIONS_OF delete_segment(as, al, sn, al);
    ASSERTIONS
       h_as_exists(as, al);
    EFFECTS
       EFFECTS_OF delete_segment(as, al, sn, al);

END_MODULE
```

```
MODULE system_io


    PARAMETERS

BOOLEAN h_device_exists(INTEGER dev_ind)  $( true for existing
                                             I/O device);


    EXTERNALREFS


    FROM dispatcher:
event: DESIGNATOR;
OFUN occurrence(event e) $( Event e has occurred) ;


    FUNCTIONS


VFUN h_device_event(INTEGER dev_ind) -> event e;
          $( Returns the event which can occur when device status
           changes)
    HIDDEN;
    INITIALLY
       e = ?;

VFUN h_input(INTEGER dev_ind) -> INTEGER data;
          $( The current input from the device)
    HIDDEN;
    INITIALLY
       data = ?;
```

```
VFUN h_output(INTEGER dev_ind) -> INTEGER data;
          $( The next output to the device)
    HIDDEN;
    INITIALLY
       data = ?;

VFUN h_command(INTEGER dev_ind) -> INTEGER command;
     $( The next command for the device)
    HIDDEN;
    INITIALLY
       command = ?;

VFUN h_status(INTEGER dev_ind) -> INTEGER status;
          $( The current status of the device)
    HIDDEN;
    INITIALLY
       status = ?;

OFUN set_event(INTEGER dev_ind; event e);
          $( When status changes event e may occur)
    EXCEPTIONS
       ~h_device_exists(dev_ind);
    EFFECTS
       'h_device_event(dev_ind) = e;

OVFUN read_device(INTEGER dev_ind) -> INTEGER data;
      $( Read data from device)
    EXCEPTIONS
       ~h_device_exists(dev_ind);
       h_input(dev_ind) = ?;
    EFFECTS
       data = h_input(dev_ind);
       'h_input(dev_ind) = ?;
```

```
OFUN write_device(INTEGER dev_ind; INTEGER data);
          $( Output data to device)
    EXCEPTIONS
       ~h_device_exists(dev_ind);
       h_output(dev_ind) ~= ?;
    EFFECTS
       'h_output(dev_ind) = data;

OFUN send_command(INTEGER dev_ind; INTEGER command);
     $( Give command to device)
    EXCEPTIONS
       ~h_device_exists(dev_ind);
       h_command(dev_ind) ~= ?;
    EFFECTS
       'h_command(dev_ind) = command;

VFUN receive_status(INTEGER dev_ind) -> INTEGER status;
     $( Get the devices status)
    EXCEPTIONS
       ~h_device_exists(dev_ind);
       h_status(dev_ind) = ?;
    DERIVATION
       h_status(dev_ind);

OVFUN device_output()[INTEGER dev_ind] -> INTEGER data;
          $( Device reads data it is to output)
    EXCEPTIONS
       h_output(dev_ind) = ?;
    EFFECTS
       data = h_output(dev_ind);
       'h_output(dev_ind) = ?;

OFUN device_input(INTEGER data)[INTEGER dev_ind];
          $( Device places input data into input buffer)
    EXCEPTIONS
```

```
        h_input(dev_ind) ~= ?;
    EFFECTS
        'h_input(dev_ind) = data;

OVFUN device_command()[INTEGER dev_ind] -> INTEGER command;
          $( The device gets a command)
    EXCEPTIONS
        h_command(dev_ind) = ?;
    EFFECTS
        command = h_command(dev_ind);
        'h_command(dev_ind) = ?;

OFUN change_status(INTEGER status; BOOLEAN occur)
                  [INTEGER dev_ind];  $( Reset the status of the
                                         device)
    EFFECTS
        'h_status(dev_ind) = status;
         ( occur AND h_device_event(dev_ind) ~= ?)
        => EFFECTS_OF occurrence(h_device_event(dev_ind));

END_MODULE
```

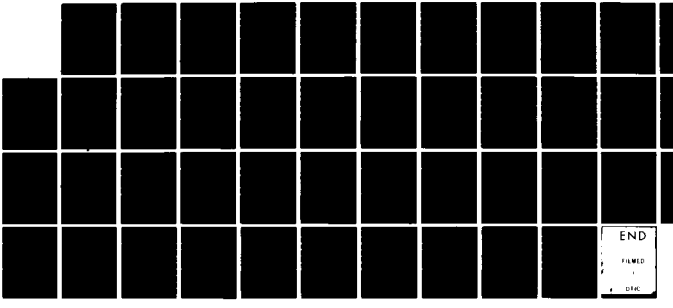
```
MODULE dispatcher


    TYPES


process, event, wakeup, processor: DESIGNATOR;
machine_state: DESIGNATOR;
program_counter: INTEGER;
process_type: { iterative, demand, background };
process_info:
STRUCT_OF(process_type type;
          INTEGER next_service;
          INTEGER interval;
          INTEGER duration;
          INTEGER processing_remaining;
          event ev;
          wakeup wk;
          program_counter pc;
          machine_state ms);


    PARAMETERS


INTEGER start_time $( time when system is initialized) ;
program_counter initial_pc $( the address of the first instruction
                              of the initial process);
machine_state initial_ms $( the initial register contents of the
                            initial process);
BOOLEAN processor_exists(processor pr)  $( true for all processors
                                           on the system);
INTEGER max_iterative $( the maximum number of iterative processes),
        max_demand $( the maximum number of demand processes),
        max_background $( the maximum number of background procs);
```

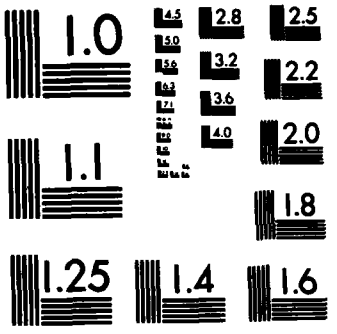
DEFINITIONS

```
BOOLEAN time_critical_process(process p)
   IS h_process_info(p).type = iterative
      OR h_process_info(p).type = demand;
BOOLEAN process_ready(process p)  $( TRUE if process p is ready to
                                     run)
   IS ( time_critical_process(p)
       => h_process_info(p).next_service <= h_time())
     AND ~(EXISTS processor pr: h_running(pr) = p)
     AND ~(p INSET h_waiting_procs());
INTEGER proc_priority(process p)  $( the scheduling priority of
                                     process p)
   IS IF p = ?
       THEN 0
       ELSE IF time_critical_process(p)
            THEN IF h_process_info(p).processing_remaining < 0
                 THEN 2
                 ELSE  3 + CARDINALITY(
                           {INTEGER i |
                             EXISTS process p1 |
                              time_critical_process(p1):
                               i = h_process_info(p1).interval
                               AND i > h_process_info(p).interval}
                                 )
            ELSE 1;
VECTOR_OF process ready_processes  $( a list of all ready processes
                                      in decreasing order of priority)
   IS SOME VECTOR_OF process rq |
       (FORALL process p | process_ready(p):
         EXISTS INTEGER i: rq[i] = p)
       AND (FORALL INTEGER i | 0 < i AND i <= LENGTH(rq):
            process_ready(rq[i]))
       AND (FORALL INTEGER i | 0 < i AND i <= LENGTH(rq):
            FORALL INTEGER j | 0 < j AND j <= LENGTH(rq):
```

AD-A122 706 TACTICAL EXECUTIVE (TACEXEC): A REAL-TIME SECURE OPERATING SYSTEM FOR TACTICAL APPLICATIONS(U) SRI INTERNATIONAL MENLO PARK CA R J FEIERTAG ET AL. JUL 79 2/2

UNCLASSIFIED DAAB07-76-C-0368 F/G 9/2 NL





```
            ( rq[i] = rq[j] => i = j)
            AND (proc_priority(rq[i]) > proc_priority(rq[j])
                 => i < j));
VECTOR_OF processor preemptable_processors
          $( a list of all processors in increasing order of the
           priority of the processes running on them)
   IS SOME VECTOR_OF processor pq |
       (FORALL processor pr | processor_exists(pr):
         EXISTS INTEGER i: pq[i] = pr)
       AND (FORALL INTEGER i | 0 < i AND i <= LENGTH(pq):
             processor_exists(pq[i]))
       AND (FORALL INTEGER i | 0 < i AND i <= LENGTH(pq):
             FORALL INTEGER j | 0 < j AND j <= LENGTH(pq):
              (pq[i] = pq[j] => i = j)
              AND (  proc_priority(h_running(pq[i]))
                    < proc_priority(h_running(pq[j]))
                   => i < j));
REAL processor_utilization(SET_OF process sp)
   IS IF sp = {}
       THEN 0.0
       ELSE LET process p | p INSET sp
             IN h_process_info(p).duration
                 / (1.0 * h_process_info(p).interval)
               + processor_utilization(sp DIFF {p});
REAL ln_2 IS 0.693 $( natural logarithm of 2 );


    FUNCTIONS

VFUN h_process_exists(process p) -> BOOLEAN b;  $( True if process
                                                   p exists)
    HIDDEN;
    INITIALLY
       CARDINALITY({ process p | h_process_exists(p) }) = 1;
```

```
VFUN h_process_info(process p) -> process_info pi;
        $( Returns scheduling information about a given process)
    HIDDEN;
    INITIALLY
        pi
       =(IF h_process_exists(p)
          THEN STRUCT(background, ?, ?, ?, ?, ?, ?,
                      initial_pc, initial_ms)
          ELSE ?);

VFUN h_running(processor pr) -> process p;
        $( Returns the process running on the given processor, or
         returns ? if the processor is idle)
    HIDDEN;
    INITIALLY
         CARDINALITY({ processor pr |
                      h_process_exists(h_running(pr)) })
        = 1
       AND( ~ h_process_exists(h_running(pr))
           => h_running(pr) = ?);

VFUN h_waiting_procs() -> SET_OF process sp;
     $( list of all waiting processes )
    HIDDEN;
    INITIALLY
       sp = {};

VFUN h_event_exists(event e) -> BOOLEAN b;  $( True if the event e
                                               exists)
    HIDDEN;
    INITIALLY
       b = FALSE;
```

```
VFUN h_wakeup_exists(wakeup w) -> BOOLEAN b;
     $( true if wakeup w has been created )
    HIDDEN;
    INITIALLY
       b = FALSE;

VFUN h_time() -> INTEGER time; $( Clock time)
    HIDDEN;
    INITIALLY
       time = start_time;

OVFUN create_process_identifier() -> process p;  $( Creates a new
                                                    token for a
                                                    process)
    EFFECTS
       ~ h_process_exists(p);
       'h_process_exists(p);

OVFUN create_event() -> event e; $( Create a new event type)
    EFFECTS
       ~ h_event_exists(e);
       'h_event_exists(e);

OVFUN create_wakeup() -> wakeup w;
      $( creates a new wakeup type )
    EFFECTS
       ~h_wakeup_exists(w);
       'h_wakeup_exists(w);

OFUN block(BOOLEAN awaiting_wakeup)[processor pr];
     $( The process running on processor pr wishes
        to relinquish the processor)
    DEFINITIONS
       process p IS h_running(pr);
    EXCEPTIONS
```

```
        awaiting_wakeup AND h_process_info(p).wk = ?;
        ~time_critical_process(p) AND ~awaiting_wakeup;
    EFFECTS
        'h_running(pr) = ready_processes[1];
        IF awaiting_wakeup
         THEN 'h_waiting_procs() = h_waiting_procs() UNION {p}
         ELSE
          (h_process_info(p).type = iterative
            => 'h_process_info(p).next_service > h_time()
               AND  'h_process_info(p).next_service
                    - h_process_info(p).interval
                   < h_time()
               AND (EXISTS INTEGER n:
                        h_process_info(p).next_service
                      + n * h_process_info(p).interval
                      = 'h_process_info(p).next_service))
          AND (h_process_info(p).type = demand
                => 'h_process_info(p).next_service = ?)
          AND ('h_process_info(p).processing_remaining
                = h_process_info(p).duration);

OFUN tick(); $( Time passes so increment the clock)
    EFFECTS
        FORALL INTEGER i |
         0 < i
         AND i <= MIN({ LENGTH(ready_processes),
                        LENGTH(preemptable_processors) })
         AND proc_priority(ready_processes[i])
             > proc_priority(h_running(preemptable_processors[i])):
          'h_running(preemptable_processors[i]) = ready_processes[i];
        'h_time() = h_time() + 1;
        FORALL processor pr |
         processor_exists(pr)
         AND time_critical_process(h_running(pr)):
          'h_process_info(h_running(pr)).processing_remaining
```

```
        = h_process_info(h_running(pr)).processing_remaining - 1;
      FORALL process p | p INSET h_waiting_procs()
                         AND time_critical_process(p)
                         AND proc_priority(p)
                              > proc_priority(
                                 h_running(
                                  preemptable_processors[1])):
       'h_process_info(p).processing_remaining
        = h_process_info(p).processing_remaining - 1;

OFUN schedule_iterative_process(process p;
                                INTEGER int;
                                INTEGER dur;
                                INTEGER begin_time;
                                program_counter p_c;
                                machine_state m_s);
        $( Permit the process p to be scheduled and run as an
         iterative pro ess)
    EXCEPTIONS
       ~h_process_exists(p);
       int < dur;
       dur < 0;
       begin_time < h_time();
       CARDINALITY({process p | h_process_info(p).type = iterative})
        >= max_iterative;
    ASSERTIONS
       processor_utilization({process p | time_critical_process(p)})
         + dur / (1.0 * int)
        < ln_2 * CARDINALITY({processor pr | processor_exists(pr)});
    EFFECTS
        'h_process_info(p)
       = STRUCT(iterative, begin_time, int, dur, dur, ?,
                ?, p_c, m_s);
```

```
OPUN schedule_demand_process(process p;
                             INTEGER min_period;
                             INTEGER dur;
                             event e;
                             program_counter p_c;
                             machine_state m_s);
     $( Permit the process p to run whenever event e occurs)
    EXCEPTIONS
       ~h_process_exists(p);
       min_period < dur;
       dur < 0;
       CARDINALITY({process p | h_process_info(p).type = demand})
        >= max_demand;
    ASSERTIONS
       processor_utilization({process p | time_critical_process(p)})
         + dur / (1.0 * min_period)
        < ln_2 * CARDINALITY({processor pr | processor_exists(pr)});
    EFFECTS
        'h_process_info(p)
       = STRUCT(demand, ?, min_period, dur, dur, e, ?, p_c, m_s);

OPUN schedule_background_process(process p; program_counter p_c;
                                 machine_state m_s);
          $( Permit process p to run whenever there is time
           available)
    EXCEPTIONS
       ~h_process_exists(p);
       CARDINALITY({process p | h_process_info(p).type = background})
        >= max_background;
    EFFECTS
        'h_process_info(p)
       = STRUCT(background, ?, ?, ?, ?, ?, ?, p_c, m_s);
```

```
OVFUN stop_process(process p)
        -> STRUCT_OF(program_counter pc; machine_state ms) state;
       $(stop scheduling the given process p)
    EXCEPTIONS
       ~h_process_exists(p);
       h_process_info(p) = ?;
    EFFECTS
       'h_process_info(p) = ?;
       state = STRUCT(h_process_info(p).pc, h_process_info(p).ms);

OFUN occurrence(event e);  $( Wake up processes waiting for event e)
    EFFECTS
       FORALL process p | h_process_info(p).ev = e:
          'h_process_info(p).next_service = h_time();

OFUN wait(wakeup w)[processor pr];
     $( tell system that this process wants to wait for wakeup w )
    EXCEPTIONS
       ~h_wakeup_exists(w);
    EFFECTS
       'h_process_info(h_running(pr)).wk = w;

OFUN continue()[processor pr];
     $( this process no longer wants to wait )
    EFFECTS
       'h_process_info(h_running(pr)).wk = ?;

OFUN notify(wakeup w);
     $( wake up all processes waiting for wakeup w )
    EXCEPTIONS
       ~h_wakeup_exists(w);
    EFFECTS
       LET process p | h_process_info(p).wk = w
        IN p ~= ? => h_process_info(p).wk = ?
                     AND 'h_waiting_procs()
```

```
= h_waiting_procs() DIFF {p};

END_MODULE
```

Appendix B

Specifications of Message System

Appendix B

Specifications of Message System

```
MODULE message_system
          $( provides a primitive system for the transmittal of
           messages (vectors of words) between named users. The
           transmission adheres to the multilevel security rules)


    TYPES

user_id: DESIGNATOR;
clearance: { INTEGER i | 0 < i AND i <= max_clearance };
category_set:
{ VECTOR_OF BOOLEAN cs | LENGTH(cs) = number_of_categories };
access_level:
STRUCT_OF(clearance security_clearance;
          category_set security_categories;
          clearance integrity_clearance;
          category_set integrity_categories);
segment_number: { INTEGER sn | 0 <= sn AND sn <= max_seg_size };
offset: { INTEGER i | 0 <= i AND i <= max_seg_size };
name: { VECTOR_OF CHAR n | LENGTH(n) <= name_length };
msg_word: ONE_OF(INTEGER, name);


    PARAMETERS

INTEGER max_clearance $( the highest clearance) ,
       number_of_categories,
       name_length $( the number of characters in a name) ,
       max_users(access_level al)  $( the most users permitted at a
                                     level),
       max_msg_size(access_level al)  $( the amount of memory that
                                        can be consumed by a user at
                                        a level for messages),
       max_messages(access_level al)  $( maximum number of messages
                                        that a user at given level
                                        can receive);
user_id sec_officer $( designator for security officer) ;
name name_sec_officer $( name for security officer) ;


    DEFINITIONS
```

```
BOOLEAN read_allowed(access_level subject_al, object_al)
        IS   subject_al.security_clearance
           >= object_al.security_clearance
          AND subject_al.integrity_clearance
             <= object_al.integrity_clearance
         AND(FORALL INTEGER i | 0 < i AND i <= number_of_categories:
                ( object_al.security_categories[i]
                 => subject_al.security_categories[i])
               AND( subject_al.integrity_categories[i]
                  => object_al.integrity_categories[i]));
      BOOLEAN write_allowed(access_level subject_al, object_al)
         IS read_allowed(object_al, subject_al);
      BOOLEAN authorized_level(user_id usid; access_level al)
         IS EXISTS name n; access_level al1 | read_allowed(al1, al):
              user_exists(usid, n, al1);
      access_level al0
         IS STRUCT(1,
                   VECTOR(FOR i FROM 1 TO number_of_categories: FALSE),
                   max_clearance,
                   VECTOR(FOR i FROM 1 TO number_of_categories: TRUE))
                $( the lowest access level);


          EXTERNALREFS


          FROM virtual_memory:
      INTEGER max_seg_size;
      address_space, segment: DESIGNATOR;
      VFUN h_as_entry_owned(address_space as;
                            segment_number sn;
                            access_level al)
                           -> BOOLEAN b;
      VFUN h_as_entry(address_space as;
                      segment_number sn;
                      access_level al)
                     -> segment s;
      VFUN h_read(segment s; offset i; access_level al) -> msg_word c;


          ASSERTIONS

      FORALL access_level al: max_users(al) > 0;


          FUNCTIONS


      VFUN user_exists(user_id usid; name n; access_level al)
                      -> BOOLEAN b;  $( True if user with id and name
                                        exists at level al)
          HIDDEN;
```

```
    INITIALLY
        (FORALL access_level al:
            user_exists(sec_officer, name_sec_officer, al))
       AND(FORALL user_id usid1 ~= sec_officer;
                  name n1 ~= name_sec_officer;
                  access_level al1:
                user_exists(usid1, n1, al1) = ?);

VFUN user_ever_existed(user_id usid; access_level al) -> BOOLEAN b;
          $( TRUE if a user ever existed at a given level)
    HIDDEN;
    INITIALLY
        (FORALL access_level al:
            user_ever_existed(sec_officer, al))
       AND(FORALL user_id usid1 ~= sec_officer; access_level al:
              user_ever_existed(usid1, al) = ?);

VFUN msg_contents(user_id usid; access_level al; INTEGER i)
                  -> VECTOR_OF msg_word mw;
          $( contents of the i-th message sent to user usid at
           level al. The message is a vector of msg-word)
    HIDDEN;
    INITIALLY
       mw = ?;

OFUN create_user(name n)[user_id usid; access_level al];
          $( invoked by the security officer, presenting the proper
           user_id, to create a new user with name n, who can
           operate at all levels not exceeding al)
    EXCEPTIONS
       not_security_officer: usid ~= sec_officer;
       too_many_users: EXISTS access_level al1 | read_allowed(al,
                                                              al1):
                            CARDINALITY({ name n1 |
EXISTS user_id usid1: user_exists(usid1, n1, al1) })
                           >= max_users(al1);
        duplicate_name: EXISTS user_id usid1:
                           user_exists(usid1, n, al0);
    EFFECTS
       EXISTS user_id usid1:
          FORALL access_level al1 | read_allowed(al, al1):
               user_ever_existed(usid1, al1) = ?
              AND 'user_ever_existed(usid1, al1) = TRUE
             AND 'user_exists(usid1, n, al1) = TRUE;

OFUN delete_user(name n1)[user_id usid];
          $( invoked by the security officer to delete a user n,
           including all of his mailboxes at all levels)
    EXCEPTIONS
       not_security_officer: usid ~= sec_officer;
       no_user: FORALL access_level al; user_id usid1:
                   user_exists(usid1, n1, al) = ?;
```

```
    EFFECTS
       LET user_id usid1 | EXISTS access_level al:
                              user_exists(usid1, n1, al)
        IN FORALL access_level al1:
             'user_exists(usid1, n1, al1) = ?;

OFUN snd_msg(segment_number sn; name n; access_level al1)
           [user_id usid; address_space as; access_level al];
         $( called by a user operating at al in address space as
          to send a message to user n at al1. The message is the
          contents of segment sn)
    DEFINITIONS
       user_id usid3
          IS SOME user_id usid2 | user_exists(usid2, n, al);
       INTEGER m
          IS MAX({ INTEGER j | j ~= ?
                             AND msg_contents(usid3, al, j) ~= ? } );
       segment s IS SOME segment s1 | h_as_entry(as, sn, al) = s1;
    EXCEPTIONS
       ~ authorized_level(usid, al);
       ~ write_allowed(al, al1);
       h_as_entry_owned(as, sn, al) = ?;
       h_as_entry(as, sn, al) = ?;
        read_allowed(al, al1)
       AND(FORALL user_id usid2: user_exists(usid2, n, al1) = ?);
        read_allowed(al, al1)
       AND CARDINALITY({ INTEGER i | msg_contents(usid3, al, i)
                                     ~= ? })
          >= max_messages(al);
        read_allowed(al, al1)
       AND (SUM(VECTOR(FOR i FROM 1 TO m
                       : LENGTH(msg_contents(usid3, al, i)))))
          >= max_msg_size(al);
    EFFECTS
          (EXISTS user_id usid1: user_exists(usid1, n, al1))
        AND( CARDINALITY({ INTEGER i | msg_contents(usid3, al, i)
                                       ~= ? })
             >= max_messages(al))
        AND (SUM(VECTOR(FOR i FROM 1 TO m
                        : LENGTH(msg_contents(usid3, al, i)))))
           >= max_msg_size(al)
       => 'msg_contents(usid3, al, m + 1)
         = VECTOR(FOR i
                  FROM 1
                  TO MAX({ INTEGER j | h_read(s, j, al) ~= ? })
                : h_read(s, i, al));

VFUN read_msg(INTEGER i)[user_id usid; access_level al]
            -> VECTOR_OF msg_word mw;  $( allows a user to read
                                          the i-th message in his
                                          mail box at al)
    EXCEPTIONS
```

```
      user_is_deleted: FORALL name n:
                           user_exists(usid, n, al) = ?;
      no_message: msg_contents(usid, al, i) = ?;
   DERIVATION
      msg_contents(usid, al, i);

OFUN delete_msg(INTEGER i)[user_id usid; access_level al];
        $( allows a user to delete the i-th message in his mail
         box at al)
   EXCEPTIONS
      user_is_deleted: FORALL name n:
                           user_exists(usid, n, al) = ?;
      no_message: msg_contents(usid, al, i) = ?;
   EFFECTS
      FORALL INTEGER j:
          'msg_contents(usid, al, j)
         =(IF j INSET { 0 .. i - 1 }
             THEN msg_contents(usid, al, j)
             ELSE msg_contents(usid, al, j + 1));

END_MODULE
```

Appendix C

MULTILEVEL SECURITY RULES

Appendix C

MULTILEVEL SECURITY RULES

1. General model

A system consists of a collection of operations or functions. Each function may be invoked by a user of the system (actually the function is invoked as part of a program running on behalf of a user). When invoked, a function may take a set of arguments. A function together with a particular set of arguments is termed a function reference. When a function reference is invoked, it can cause the state of the system to change and/or return information to its invoker. The set of all function references of a system is called F and some member of this set is denoted by f.

We also define a set of security and integrity levels L. The security and integrity levels L are partially ordered by the relation "<". Multilevel security involving classifications and categories is but one example of a partial ordering of security and integrity levels, so we will be dealing here with a more general case. There are functions K and I whose domain is F and whose range is L. The functions K and I return respectively the security and integrity levels of their argument. A process is assigned a security level and an integrity level for its lifetime and may only invoke function references at these levels. (Note that a user may have several processes operating on his behalf simultaneously, and may therefore operate at several security and integrity levels.)

Finally, we introduce the relation "-->" on function references. We say that

$$f_1 \text{-->} f_2$$

(read as f_1 transmits information to f_2) if there is any possibility that the information returned by an invocation of f_2 could have been in any way effected by a prior invocation of f_1. In other words, there is some transmission of information from f_1 to f_2.

The definition of multilevel security can now be stated simply. For any f_1 and f_2 in F:

$$f_1 \text{-->} f_2 \implies K(f_1) <= K(f_2) \text{ AND } I(f_1) >= I(f_2) \qquad \text{(P1)}$$

This simply states that if there is any possibility of information transmission between two function references, then the transmitting function reference must have a security level less than or equal to the that of receiving function reference, and the receiving function reference must have an integrity level less than or equal to that of the transmitting function reference.

In other words, information can only flow upward in security or remain at the same level. An alternative definition is given in [3]. Similarly, information can only flow downward in integrity or remain at the same level.

Unfortunately, the abstract nature of this definition makes it difficult to relate to constructs used in expressing system designs. This gap can be bridged by formulating a slightly more restrictive model in less abstract terms.

2. Restricted Model

Each state variable v contains some of the state information of the system. The state variables together completely describe the state of the system. The value of each state variable may be modified by invocation of some function reference. Each state variable is assigned a security level and an integrity level which is determined by extending the functions K and I to apply to state variables as well as function references, therefore, K(v) is the security level of state variable v and I(v) is the integrity level of state variable v. The relation $-\underline{f}>$ relates two state variables such that

$$v_1 \ -\underline{f}> \ v_2$$

means that an invocation of function reference f may cause the value of v_2 to change in a manner dependent upon the previous value of v_1. In other words there is an information flow from v_1 to v_2 caused by the invocation of f. Two predicates must also be defined: the prefix form of $-\underline{f}>$

$$-\underline{f}> \ v$$

means that an invocation of the function reference f may cause the value of state variable v to change; the postfix form

$$v \ -\underline{f}>$$

means that the value returned by function reference f is dependent on the prior value of state variable v. Note that for any f_1, v_1, and v_2:

$$v_1 \ -\underline{f}> \ v_2 \ ==> \ -\underline{f}> \ v_2$$

A multilevel secure system may now be redefined. For any function reference f and state variables v, v_1, and v_2

P2

$$v \ -\underline{f}> \ ==> \ K(v) <= K(f) \ \text{AND} \ I(v) >= I(f) \qquad \text{(P2a)}$$

$$v_1 \ -\underline{f}> \ v_2 \ ==> \ K(v_1) <= K(v_2) \ \text{AND} \ I(v_1) >= I(v_2) \qquad \text{(P2b)}$$

$$-\underline{f}> \ v \ ==> \ K(f) <= K(v) \ \text{AND} \ I(f) >= I(v) \qquad \text{(P2c)}$$

These properties assure that information flow is always upward in security level, downward in integrity level, or remains at the same security or integrity level. Loosely speaking, the arrow --> always points upward in security level and downward in integrity level. P2a states that the value returned by an invocation of a function reference at some security and integrity levels contains information from state variables at only lower or equal security levels or higher or equal integrity levels. P2b assures that when information is transferred from one state variable to another by some invocation of a function reference, that the recipient variable is at a higher or equal security level or lower or equal integrity level than the originator variable. P2c assures that the value of a state variable may be changed by invocation of a function reference whose security level is less than or equal to or whose integrity level is greater than or equal to that of the variable, thereby guaranteeing that security cannot be violated by the act of invoking a function reference. An alternative definition is given in [3].

3. Formal Definitions of Relations and Predicates

A multilevel system is defined to be the following ordered 10-tuple:

$$\langle S, s_0, L, "<", F, K, I, R, N_r, N_s \rangle$$

where the elements of the system can be intuitively interpreted as follows:

- S - States: the set of states of the system
- s_0 - Initial state: the initial state of the system; $s_0 \in S$
- L - Security levels: the set of security levels of the system
- "<" - Security relation: a relation on the elements of L that partially orders the elements of L
- F - Visible function references: the set of all the externally visible functions and operations (i.e., functions and operations that can be invoked by programs outside the system); if a function or operation requires arguments, then each function together with each possible set of arguments is a separate element of F (note that in this document externally visible functions and operations will be referred to collectively as visible functions (or functions) even though operations are not functions in the mathematical sense)
- K - Function reference security level: a function from F to L giving the security level associated with each visible function reference; a process may invoke only function references at the security level of the process; K:F->L
- I - Function reference integrity level: a function from F to L giving the integrity level associated with each visible function reference; a process may invoke only function references at the integrity level of the process; I:F->L

R - Results: the set of possible values of the visible function references

N_r,N_s - Interpreter: functions from FXS to R and S that define how a given visible function reference invoked when the system is in given state produces a result and a new state; N_r:FXS->R and N_s:FXS->S.

There is also a set of state variables V, each member of which is the set of values can be assumed by that state variable. The set of states S is isomorphic to the cross product of all the state variables v∈V.

In order to define multilevel security and integrity, the following definitions are useful:

T - the set of all n-tuples of visible function references or, in other words, all possible sequences of operations

$T = F^*$

M - the function whose value is the state resulting from the given sequence of operations starting at some given state

M:SXT->S

D - the function whose value is the set of state variables whose values differ in the given states

$D:SXS->V^*$

The two relations and two predicates described above can now be formally defined:

f_1 --> f_2 <==>

$(\exists t_1,t_2 \in T)$

$N_r(f_2,M(t_2,M(\langle f_1\rangle,M(t_1,s_0))))$

$\tilde{=} N_r(f_2,M(t_2,M(t_1,s_0)))$

$v_1 \xrightarrow{f} v_2$ <==>

$(\exists s_1,s_2 \in S \mid D(s_1,s_2)=\{v_1\})$

$v_2 \in D(N_s(f,s_1),N_s(f,s_2))$

$v \xrightarrow{f}$ <==>

$(\exists s_1,s_2 \in S \mid D(s_1,s_2)=\{v\})$

$N_r(f,s_1) \tilde{=} N_r(f,s_2)$

$\xrightarrow{f} v$ <==>

$(\exists s \in S)$

$v \in D(s,N_s(f,s))$

Appendix D

SAMPLE MULTILEVEL SECURITY PROOF

Appendix D

SAMPLE MULTILEVEL SECURITY PROOF

The actual state of the system is described by the "primitive" V-functions, i. e., functions that return the value of a particular state variable of the system. The primitive V-functions are descriptive artifacts of the specifications and need not be present in an implementation. The value of a primitive V-function may be available to a user of the system if there is a visible V-function that returns the value of the primitive V-function. The values returned by visible V-functions are functions of the values of only the primitive V-functions.

The specification of each visible function has two major parts. The first part is the EXCEPTIONS, a list of boolean valued expressions. If any of these expressions evaluates to true for a given invocation of a function, then the function is aborted with no change of state to the system. The values of these exceptions are results of the function invocation since the occurrence of an exception is reported to the caller of the aborted function.

For a visible V-function, the second part of the function specification is the DERIVATION, an expression whose value is the result of the V-function. The value is returned only if all the exceptions of the V-function invocation are false. For an O- or OV-function, the exceptions are followed by the EFFECTS, assertions that relate the values of the state variables (primitive V-function references) subsequent to the invocation of the OV-function to the values of the state variables prior to the invocation of that OV-function. Subsequent values of state variables are denoted in effects by preceding the primitive V-function references corresponding to those state variables by a single quote ('). Prior values are unquoted.

Note that there is a very strong correlation between the model underlying the semantics of SPECIAL and the model of a system used to describe the strong multilevel security properties, P2. The state variables, V, of the security model are references of the primitive V-functions of SPECIAL and the function references, F, of the security model are references of the visible functions of SPECIAL. The values of function references of the security model are the return values and exceptions of the visible functions in SPECIAL. We have also added a convention that prescribes that each primitive function reference of a SPECIAL specification contain a formal parameter that is the security and integrity levels of that function reference. For visible V-functions, the security and integrity levels of a function reference are implicit arguments enclosed in square brackets ([...]) after the formal parameter list. The properties P2a, P2b, and P2c can, therefore, be directly applied to specifications written in SPECIAL.

There are two difficulties that make proof of the consistency of the specifications and the properties P2 nontrivial. First, the specifications are written in terms of function descriptions, not function reference descriptions. This means that one must prove that the properties P2 hold for all possible arguments to the functions described in the specifications. In many cases some sets of arguments to a particular function must be considered as distinct cases in order to make the proof tractable. The appropriate partitioning of cases requires careful judgment. Second, in describing the change of state caused by an O- or OV-function invocation, SPECIAL permits considerable freedom in expressing the relation between the new values of the primitive V-function references and their prior values. The use of recursive functions and universal and existential quantifiers makes it undecidable, in general, to determine if a new value of a primitive V-function reference is functionally dependent upon the prior value of some other primitive V-function reference. Since functional dependency is generally undecidable, we have derived a set of decidable dependency rules that are used to determine if the value of some quoted primitive V-function reference (new value of a state variable) is functionally dependent upon some unquoted primitive V-function reference (prior value of a state variable) for the most common of the decidable cases. When these rules cannot be definitively applied, a functional dependency is assumed. These rules are similar to the elimination rules of [4]. For the specifications we have examined, we have had no difficulty in deriving an acceptable set of such rules. The example given later illustrates the proof technique and utilizes a particularly simple set of these decidable dependency rules. In order to illustrate the proof technique, a proof of two representative operations will be presented. The operations whose security will be demonstrated are SEGMENT_READ and GET_SEGMENT in the module VIRTUAL_MEMORY. These operations may be considered representative in style, size, and complexity of operations in the TACEXEL design, being perhaps a little simpler than most. The proof of properties P2a, P2b, and P2c require the identification of all instances of primitive V-function references within the operation to be proved. Many such instances are enclosed in the macro facilities of SPECIAL (namely the DEFINITIONS, EXCEPTIONS_OF, and EFFECTS_OF) so those macro definitions containing primitive V-function references must be expanded. However, no such expansions are necessary for the sample operations.

Each function reference must be assigned a security and integrity level, collectively called an access level. In order to guarantee that the levels of function references do not change (a requirement of the multilevel model), one of the arguments to each function reference will be its access level. By convention, the access level argument will be the formal parameter in the definition of the function that is named "al". (Note that this choice of access level is arbitrary; an incorrect choice may cause the proof to fail, however it is never possible to make a choice that will cause the proof to succeed for an insecure system.) The relation "<=" is defined for access levels by the definition "write_allowed" and the relation ">=" is defined for access level by the definition "read_allowed".

The next step in the proof process is to generate a set of theorems whose validity implies properties P2a, P2b, and P2c. These theorems are derived from the specifications using knowledge of the syntax of SPECIAL and the decidable dependency rules (which embody the semantics of SPECIAL). An examination of these theorems serves to illustrate the theorem-generating step of the proof process. The theorem for the operation SEGMENT_READ is:

FORALL sn, i, as, al: read_allowed(al, al)

Properties P2a, P2b, and P2c must be proved for each visible function reference, i.e., the proof must be carried out for all possible set of arguments, hence the universal quantification of all the arguments in the theorem. In actuality, six theorems are generated from SEGMENT_READ, one for each primitive V-function reference, but they are all identical.

Consider first the exceptions. Recall that the value of an exception is a result of a visible function, so it is necessary to identify all primitive V-function references (state variables) upon which the values of the exceptions are dependent and prove that their levels are less than or equal to level of SEGMENT_READ. All the primitive V-function references in the exceptions are at level "al" and the level of SEGMENT_READ is also "al", so we must prove that al<=al. This is the what the theorem above expresses and it is obviously true from the definition of read_allowed.

Next we must consider the derivation of SEGMENT_READ. The value of the derivation is also a result of the function, so again we must identify all the primitive V-functions in the derivation and prove that their level is less than or equal to the level of SEGMENT_READ. Again all the primitive V-functions have level "al" and the same true theorem obtains. Since this operation is a V-functions and there is no change of state, properties P2b and P2c do not apply. This completes the proof of SEGMENT_READ.

The proof of SEGMENT_READ is fairly simple because all the primitive V-functions referenced in SEGMENT_READ are at the same security level. We did not even consider if an exception or result is dependent upon a particular primitive V-function reference, we simply assumed the worse case that if a primitive V-function appeared in an expression, then the value of the expression was dependent on the value of that primitive V-function reference. In the next example, GET_SEGMENT, the proof is not so simple and it is necessary to consider more carefully whether or not there is a transmission of information from some primitive V-function reference.

Consider first the first exception of GET_SEGMENT. In this exception there is a reference to H_AS_EXISTS. Note however, that the value of the exception is dependent on the value of the reference to H_AS_EXISTS only if the read_allowed(al, asl) is true. If read_allowed(al, asl) is false then the exception is false no matter what the value of the reference to H_AS_EXISTS. This leads to the following theorem:

```
FORALL source_as, asl, source_sn, dest_sn, as, al:
 read_allowed(al, asl) => read_allowed(al, asl)
```

The consequent of the implication is the normal check for the security level of the function being higher than the security level of the state variable (property P2a). The antecedent of the implication qualifies the check for those cases where it matters. This theorem is trivially true. The second exception yields the same theorem for the same reason.

The third exception yields the following theorem in the normal manner:

```
FORALL source_as, asl, source_sn, dest_sn, as, al:
 read_allowed(al, al)
```

which is true from the definition of read_allowed.

Now consider the first effect of GET_SEGMENT. In order to demonstrate property P2b we must show that the level of the modified primitive V-function reference to H_AS_ENTRY is greater than that of the unmodified primitive V-function reference to H_AS_ENTRY. Again we need consider only the cases in which there is actually transmission of information, so the following theorem results:

```
FORALL source_as, asl, source_sn, dest_sn, as, al:
 FORALL l:
   read_allowed(l, asl) AND read_allowed(l, al)
  => read_allowed(l, asl)
```

which is trivially true.

To show property P2c we must prove that the level of all modified primitive V-function references is greater than the level of the reference to GET_SEGMENT. There are modified primitive V-function references to H_AS_ENTRY and H_AS_ENTRY_OWNED. For H_AS_ENTRY we consider only the relevant cases yielding:

```
FORALL source_as, asl, source_sn, dest_sn, as, al:
 FORALL l:
   read_allowed(l, as) AND read_allowed(l, al)
  => read_allowed(l, al)
```

which is trivially true. For H_AS_ENTRY_OWNED the resulting theorem is:

```
FORALL source_as, asl, source_sn, dest_sn, as, al:
 read_allowed(al, al)
```

which is true from the definition of read_allowed.

A simple upper bound can be placed on the number of theorems generated for a given visible function. Using the following definitions:

nxv = the number of citations of primitive V-functions in the exceptions

nqv = the number of citations of quoted primitive V-functions in the effects

nuv = the number of citations of unquoted primitive V-functions in the effects or derivation

the number of theorems generated will be at most

$$nxv + (nqv + 1) * nuv + nqv$$

For the SEGMENT_READ operation this upper bound is 6 and for the GET_SEGMENT operation this upper bound is 8. In these cases the failure to reach the upper bound is due to the absence of a return value (other than the exceptions) and that some of the theorems happen to be identical and have not been replicated.

It is important to realize that this particular example is probably smaller than the proof of a typical visible function in a system such as TACEXEL. A more representative example is likely to contain more DEFINITIONS, EXCEPTIONS_OF, and EFFECTS_OF expressions that contain citations of primitive V-functions thereby yielding a much greater number of such citations in the expanded form of the function specification, hence a much greater number of theorems. In fact, the listing of theorems is undoubtedly going to be much longer than the listing of the specifications from which the theorems are derived. The saving grace is that the proofs of the theorems are rather simple and are amenable to automation.

Appendix E

PERMISSIBLE PROCESSOR LOADINGS

Appendix E

PERMISSIBLE PROCESSOR LOADINGS

This appendix examines three scheduling algorithms to determine the processor loads which can be sustained without risk that any task cannot be serviced within its time constraints. Subsequent work should consider the effects of scheduling overheads and extend the analysis to other workloads and scheduling algorithms.

1. Deadline Scheduling

For this analysis nothing is assumed about the nature of the tasks to be performed except that:

a) for each task there is a known deadline by which the task must be completed,

b) the processing of any task may be preempted should it be appropriate to process another task,

c) the tasks may be processed at any time between their initiation and their deadline, and thus tasks may not block each other by semaphores or other mechanism.

Scheduling overheads are assumed to be zero. The analysis makes no assumptions about the periodic repetition of tasks, about foreknowledge of the processing requirements of tasks, or about the possible future demands of other tasks.

The deadline scheduling algorithm to be analysed selects, from amongst those tasks available for processing, that task whose deadline is the earliest. It will be shown that if, for any particular combination of tasks, the deadline scheduling algorithm is unable to schedule tasks so as to complete all of them within their respective deadlines, then there does not exist any schedule which is able to complete them all.

Consider a pattern of tasks such that one task cannot be completed before its deadline. From this pattern, select a critical subset of tasks by recursive enumeration, using the rules:

a) the task which cannot be completed within its deadline is a member of the set,

b) any task, such that at the time at which its processing is completed some other task of the set awaits service, is a member of the set.

Note that during the interval from the earliest initiation of a member of the set to the deadline of the failing task there must always be at least one member of the set being processed or awaiting processing.

Thus within this interval there can be no idle time and no processing performed for any task whose deadline is later than that which was failed.

Consider this interval. The set of tasks is a set all of whose processing must be completed within the interval. The whole of the interval is allocated to processing for these tasks, and yet processing remained at the end of the interval. Thus the total quantity of processing required during the interval exceeds the length of the interval, and there can be no arrangement of processing which can complete it in time.

Thus if, for any particular combination of tasks, the deadline scheduling algorithm is unable to schedule tasks so as to complete all of them within their respective deadlines, then there does not exist any schedule which is able to complete them all within their respective deadlines The corollary of course is that if there exists any schedule which can complete all the tasks, then the deadline schedling algorithm suffices to find such a schedule.

2. Priority Scheduling of Periodic Tasks

For this analysis, it is assumed that the tasks of the system are activated on a regular periodic basis, and that each task must complete its processing before the next activation of that task is due. It is assumed that tasks may be scheduled to be run at any time within this period, that preemption is permitted, and that the scheduling overhead is zero.

The priority scheduling algorithm to be analysed selects, from amongst those tasks available for processing, that task whose repetition period is shortest. It will be shown that a particular pattern of task periods and activations represents a local most difficult case and that scheduling on the basis of repetition period permits the highest loading for this pattern. It is shown that for this local worst case the processor may be loaded upto ln(2) of capacity without risk of any task failing to complete within its repetition period.

It is believed that the bad patterns of tasks occur when all the tasks of the system are activated at the same moment in time, and when, for each task except that with the shortest period, the period of a task is equal to its own processing time plus the period of the next shorter period task. An example of such a bad pattern is given in Figure E-1. Contrary to intuition, the pattern in which all tasks must complete before the same moment in time, shown in Figure E-2, is not a bad pattern. Figure E-3 shows that the deadline algorithm described above can schedule a bad pattern for which the priority algorithm can find no schedule.

It is believed that the worst case pattern occurs when the processing times for all tasks are in equal proportion to their periods. While this proportion tends to zero as the number of tasks in the pattern tends to infinity, no demonstration is available that this pattern is a global worst case. It can only be shown that each change

in the pattern permits a higher processor utilization, indicating that the pattern is a local worst case.

Consider a pattern similar to that of Figure E-1 with n tasks processed and task n+1 unprocessed. If the period of the shortest period task is a then its processing requirement must be $a(2^{1/n}-1)$ while the period of the next task is $a(2^{1/n})$. The period of each task in the series increases in the same proportion until the period of task n+1 is $a(2^{1/n})^n = 2a$. The processor load is

$$n(2^{1/n}-1)$$

$$= n(e^{\ln(2)/n}-1)$$

$$= n(1 + \ln(2)/n + \ln(2)^2/2n^2 + \ln(2)^3/3!n^3 + \ldots - 1)$$

$$= \ln(2) + 0(1/n)$$

Thus the permissible load decreases monotically as n increases with a limit of ln(2) (~0.693). If the worst case pattern of tasks can be processed using the priority scheduling algorithm, provided that the processor load does not exceed ln(2), then any pattern of tasks can be processed using the priority scheduling algorithm provided load does not exceed ln(2) (approximately 0.693).

3. Simply Periodic Scheduling

The bad cases for the priority scheduling of periodic tasks arise because the arbitrary periods of the periodic tasks allow the relative phasing of those tasks to change until a bad case is builtup. If the periods of the tasks are constrained to be simple multiples of each other, this effect can be avoided and higher processor utilizations can be permitted.

We define a simply periodic system to be one in which each period, other than the shortest, is an integral multiple of the next shorter period, and initialy all periods start simultaneously. Several tasks may be run at each of these periodicities. The scheduling algorithm selects at all times tasks of the shortest period which still require processing. An example of a simply periodic system is given in Figure E-4.

Simple inspection shows that provided the load on the system does not exceed the capacity of the processor, the simply periodic system can complete all tasks within their periods. This high processor utilization is, however, to some extent misleading. Because of the limited set of periodicities at which tasks may be run, some tasks may be run more frequently than the application really requires. If the required processor loading is uniform over task period, then conversion to a simply periodic system is equivalent to restricting the processor loading to 0.75. If the required processor loading is negative exponential over task period, then conversion to a simply period system is equivalent to restricting the processor loading to 0.682. In other cases, the required periods may be such that very little increase in processor load results from conversion to a simply periodic system

4. Demonstration of local worst case

Consider the pattern of Figure E-1. To demonstrate a local worst case, it is necessary to show that a change in the time of activation, or in the period, or in the processing requirement of any task loads to an increase in processor utilization over the bad case.

If any task of B to F is activated slightly early, time will be available for processing task G after task F has been processed. If task A is activated slightly early, time will be available for processing task G after task A has been processed for the second time. If task G is activated early, time is available for processing it at that time. These changes all increase processor utilization and lead away from the worst case.

If any task of A to F is activated slightly late, time will be available for processing task G before the second activation of that task. If task G is activated slightly late, the pattern is unchanged, while for greater delays in activation of task G, time is available for processing task G after the third processing of task A. These changes all increase processor utilization and lead away from the worst case.

If any task of B to F has a slightly shorter period, the pattern of usage is unchanged and the processor utilization is increased. If task A has a slightly shorter period, the processing of task F is split, and the processor utilization is increased. Thus these changes lead away from the worst case.

If any task has a slightly longer period, then time is available for the processing of task G immediately prior to the activation of that task for the second time. It is necessary to show that the increased processor utilization from processing task G is greater than the decreased processor utilisation from the longer period. Lengthening the period of task A yields the greatest reduction in processor utilisation. If the period of task A is increased by s and task G processes for s the change in processor utilization is

$$-q/nq + q/(nq + d) + d/2nq$$

$$= ((n-2)dq + d^2)/(2nq(nq + d))$$

which is positive, indicating an increase in processor utilization and a move away from the worst case.

Changes in task processing time require more care since a move towards shorter processing time and more tasks is a more towards the worst case. It is appropriate to show that a move away from processing times proportioned to task period is a move away from the worst case. Consider two tasks of period a and a+q. If the processing requirement of the short period task is reduced by d, then the period of the longer period task is reduced by d and its processing is increased by d. The change in processor utilization is

$$-q/a + (q-d)/a - q(q+a)/(a(q+a)) + (q + a)q/a$$

$$= d^2/(a(q+a-d))$$

Since this is a term in d^2 any change in the processing requirement of a

task, whether an increase or decrease, increases processor utilisation and is a move away from the worst case.

To show that a schedule based on task repetition periods is the best fixed priority schedule, consider interchange of priorities for two tasks. For any two tasks of A to F, the execution pattern changes but there is still no time in which to process task G. If task G's priority is interchanged with any task A to E that task will fail to complete its processing within its first iteration. Since G processes for longer than the task whose priority it has taken task F will also fail to complete. If tasks F and G are interchanged in priority, G will complete while F fails, and processor utilisation is reduced.

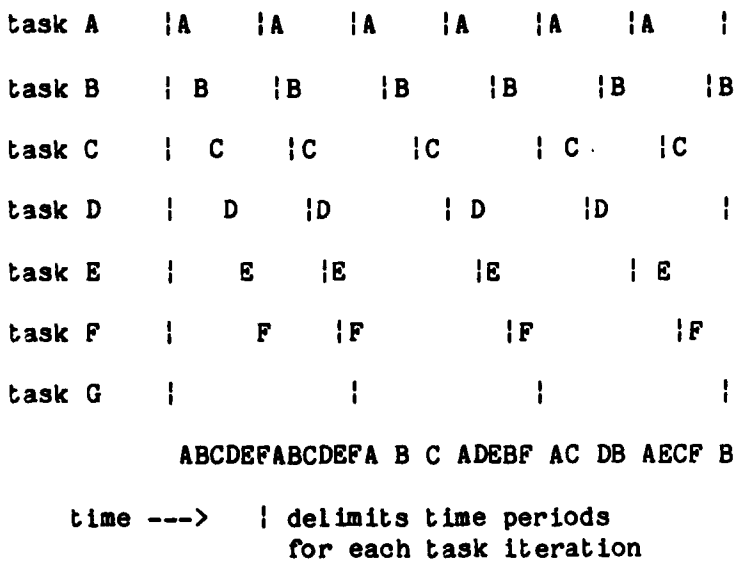


Figure E-1. A Bad Pattern of Tasks for Priority Scheduling

Note that, though the processor is not yet fully loaded, there is no time available for processing task G within its first iteration. For this task pattern the maximum safe processor load is about 0.736. As the number of tasks in the pattern increases, the maximum safe processor load diminishes to about 0.693.

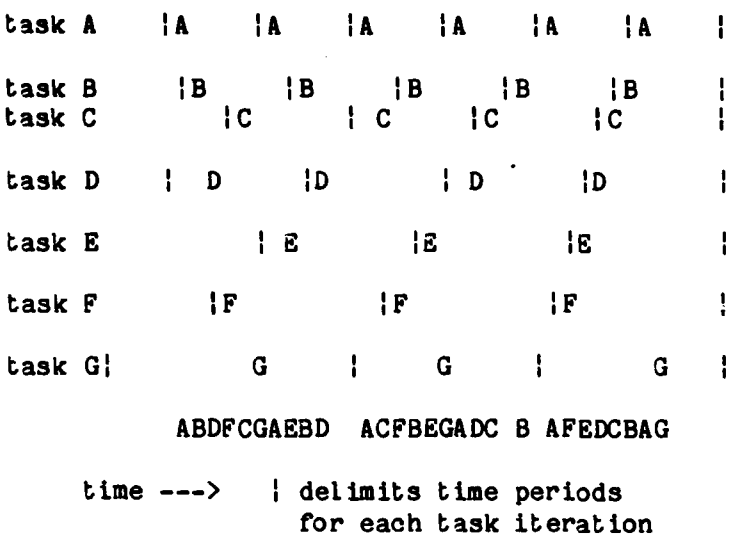


Figure E-2. Many tasks completed at the same time

Contrary to intuition this pattern presents no scheduling problems (yet; it is of course followed immediately by the pattern of Figure E-1).

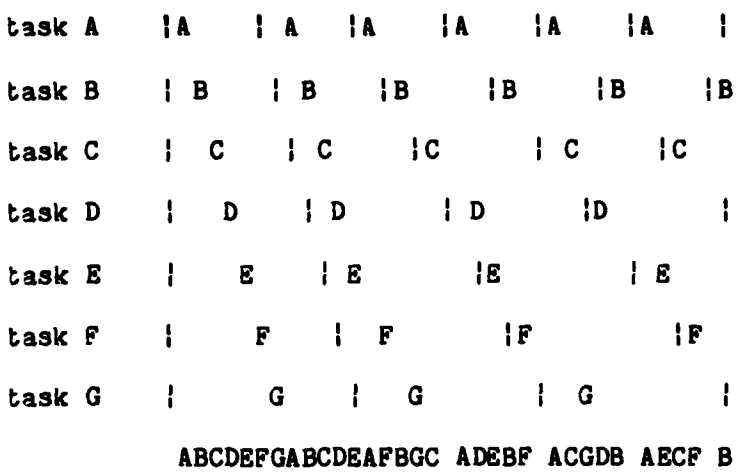


Figure 3. Tasks from Figure E-1, using the
Deadline Scheduling Algorithm

The deadline algorithm has no difficulty in running task G and could even find time to run further tasks, in a circumstance in which the priority algorithm could not run task G.

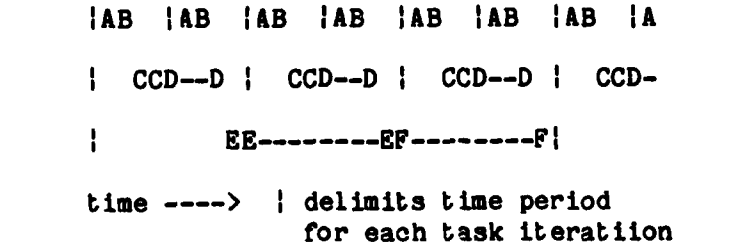


Figure E-4. A sample pattern of tasks for
a simply periodic system

Note that, provided each set of tasks operate on a period which is an integral multiple of the next shorter period, high processor utilisation can be achieved safely. If any pair of periods do not have a simple integral relationship, this high utilization cannot be permitted.

Appendix F

THE SPECIAL SPECIFICATION LANGUAGE

Appendix F

THE SPECIAL SPECIFICATION LANGUAGE

SPECIAL is a specification language used for specifying the functional behavior of modules (Stage 4) and for describing representations (Stage 5).

The language originated in the work of Parnas [7], but has evolved significantly since. SPECIAL lacks some of the mathematical elegance of the algebraic specification technique [15], but is a more powerful language capable of expressing some specifications that cannot be expressed at all by any other specification language. If the full power of SPECIAL is used, there is no hope of showing that a specification is complete and consistent,and satisfies a requirement statement, e.g. the multi-level security model. Indeed it is a feature of SPECIAL that nondeterministic systems can be specified. However few specifications need the full power of SPECIAL, and it is possible to write specifications within the kind of restricted domain that allows straight-forward derivation of the properties of the specification.

1. Description of the Language

The heart of a specification written in SPECIAL is the definition of the operations on the type. The operations are of three kinds:

- O-functions (OFUN),
- OV-functions (OVFUN),
- V-functions (VFUN).

In the absence of exceptional conditions:

- a V-function invocation (as an operation) returns a value, but causes no state change,
- an O-function invocation can cause a state change, but returns no value an OV-function invocation returns a value and can cause a state change

A V-function is denoted as visible if it is an operation of the type and as hidden if it is internal to the specification. A V-function may also be derived, meaning that its value is expressed as a function of the values of other V-functions. The "state" of the type can be thought of informally as the Cartesian product of the values of all of the V-functions other than the derived functions. Good practice in the use of SPECIAL requires that all the visible V-functions be derived, so that the state functions are all hidden.

In addition, the specification defines:

* initial values for each nonderived V-function. The specification is required to define initial values for the full domain of the V-function.
* exception conditions for each of the visible V-functions, O-functions, and OV-functions.
* the returned value for each derived V-function and OV-function.
* the values that the nonderived V-functions will acquire after an invocation of each O-and OV-function.
* assertions about relationships between the values of the parameters.

SPECIAL allows user-defined local functions. The definition of the function gives a type to the function and to each of its formal arguments, and provides a body. Any such function can be used as a sub-expression in an expression with appropriate actual arguments substituted for the formal arguments, provided the type of the actual arguments is consistent with the function definition, and the declared type of the function is consistent with its use in the expression. For example, we can define the Boolean function no-string using the following syntax

```
BOOLEAN no-string( INTEGER j) IS
    j < 1 OR j > t_len(),
```

where the body follows the reserved word IS, and t_len() is a V-function of the module. One can use no-string(i) where a Boolean-value is expected within a scope where i has been declared as an integer.

Designators

A designator is the name of an object or an instance of the type being defined. Designators are not manipulatable, except for being returned as the result of a function or being used as an argument to a function.

Sets

In specifying a concept it is often useful to view objects as if they formed a set. The advantage of the set viewpoint is the absence of any consideration of ordering or repeated elements. The use of sets in a specification often leads to simpler specifications and averts prejudicing a specification with implementation decisions. All elements of a set are of the same type.

If s has been declared to be of type

```
SET_OF INTEGER
```

then s can be defined to be a particular integer set. The extensional constructor explicitly identifies the individual elements. The following forms are equivalent:

```
s = {1, 3, 5, 7}
s = SET(1, 3, 5, 7)
```

The intentional constructor can also be used:

```
s = {INTEGER i | o < i AND i < 9 AND i MOD 2 = 1 }
```

The general form for a integer set is

```
{INTEGER i | p(i) }
```

where p(i) is a Boolean expression. The intentional form is used more often, since it permits the concise characterization of large sets.

The set of consecutive integers between two given integers can be specified using the following shorthand:

```
ss = {7 .. 36 }
```

A predefined function for sets, is CARDINALITY, which returns an integer, the number of elements in a set. Thus,

CARDINALITY(s)

would now be 4.

Another predefined function for sets is INSET, which determines that an element is in a set, returning a result of type BOOLEAN. Thus,

1 INSET s

is TRUE.

Vectors

For vectors, similar constructors are provided. If iv has been declared to be of type

VECTOR_OF INTEGER

then the extensional constructor would be used, as:

iv = VECTOR (1, 3, 5, 7)

The intentional constructor for the same vector is

iv = VECTOR(FOR i FROM 1 to 4: 2**1 - 1).

The predefined function LENGTH returns the number of elements in a vector. Thus

LENGTH(iv)

returns the integer 4.

Structures

This form is used to specify an ordered assemblage of objects, not necessarily of the same type. The elements of a structure are each identified by a unique name. The structured type employee, each value of which contains 3 elements, could be declared as follows

employee: STRUCT_OF(INTEGER id, age; VECTOR_OF CHAR title)

A particular instance, Williams, of the type employee can be expressed as

```
Williams = STRUCT(15024, 22, Sr_Adm_Aide).
```

Particular components can be referred to by using the component name as an extractor

```
Williams.age
```

has value 22.

Undefined Values

It is often useful in a specification to indicate that a particular object has no value. We use the particular symbol ? (shorthand for UNDEFINED) to represent no value. Often, the initial values of primitive V-functions are most conveniently specified to be ?, rather than some random value. In SPECIAL, ? is a member of all types unless explicitly excluded. Thus the type INTEGER consists of the values

```
{ ... , -2, -1, 0, ?, 1, ... }
```

The rules of the grammar are satisfied when a V-function is declared to be of type INTEGER, and the specification indicates that the initial value for certain of its associated V-functions is ?.

Function Definitions

A hidden V-function definition has the form:

```
VFUN v(typespec1 arg1; ... ) -> typespec result;
   HIDDEN;
   INITIALLY
      expr;
```

The expression following INITIALLY is an expression that characterizes the initial value(s) for each possible argument. Generally, "expr" is of the form

```
result = expression
```

possibly being

```
result = ?,
```

as shorthand for: result is ? in the initial state for all possible arguments to v.

A visible V-function has the form:

```
VFUN v(typespec1 arg2; ... ) -> typespec result;
   EXCEPTIONS
      ex1;
      ex2;
        .
        .
        .
   INITIALLY
      expr;
```

Each of the exception conditions is of the form

exceptionname: expression,

where "exceptionname" is name assigned to the exception condition, and expression is a Boolean expression of the arguments, V-functions, and parameters. The exceptionname enables a program using the operations of the type to discriminate between the possible exceptions. Generally, but not always, an abstract program invoking a visible function will test for the existence of the exceptions in the order they appear in the specification. Thus, if the expressions associated with d1, ... di-1 evaluate to FALSE for the arguments of the function invocation, and the expression associated with di evaluates to TRUE, then di will be "raised"; subsequent exception conditions are not tested. If "v" has no exception conditions then the "exceptions section" is omitted.

A derived V-function has the form:

```
VFUN v(typespec1 arg1; ... ) -> typespec result
   EXCEPTIONS
        .
        .
        .
   DERIVATION
      expr;
```

where the expression following DERIVATION defines the result in terms of the arguments, primitive V-functions, and parameters. The type of the expression should be the type of the function.

An OV-function has the form:

```
OVFUN ov(typespec1 arg1; ... ) -> typespec result;
   EXCEPTIONS
        .
        .
        .
   EFFECTS
      ef1;
      ef2;
        .
        .
      efq;
```

Each of the effects ef1 ... efq is an assertion that relates the value of the result and/or the new (after the invocation) value of primitive V-function positions, to the values of the arguments, the prior (before the invocation) values of V-functions, and the parameters. The notation 'v(x) is used to denote the new value of a V-function. In the EFFECTS section, the results and the new values for V-functions are defined by the conjunction of all of the effects assertions. They appear as separate expressions only for ease of presentation. There is no concept of order implied here since we could have equivalently stated the EFFECTS as the single expression

ef1 AND ... AND efq.

As indicated previously, these effects occur only when an operation does not cause any of the exception predicates to be satisfied.

The schema for an O-function is identical to that of an OV-function, except that no returned result is indicated.

With this brief introduction to SPECIAL the reader should be able to follow the example specification.

a. An Example of a Specification in SPECIAL

The module "sequences" defines a collection of word files (sequences), each of which is identified by a unique designator of type nameseg. A user of the module can request the creation of a new sequence; an existing sequence can be cleared to its initial state, but never be deleted, so that there is no recycling of nameseg designators.

For reading, the words of a sequence are randomly accessed by position. A sequence is grown by appending words to the end. Two words of a sequence can be interchanged. The operations defined are:

* nameseg; a designator type, the values of which are names of sequences.
* string (nameseg n; INTEGER j) -> word w; a visible V-function that returns the word w at position j in the designated sequence n; word is a named type that is precisely defined later. As the only V-function, string captures the "state" of each sequence in the system.
* seqlen(nameseg n) -> INTEGER v; a derived visible V-function that returns the current length of sequence n. The value of seqlen(n) can be derived from the value of string(n, j).
* create_seq() -> nameseg n; an OV-function that creates a new sequence, initializes it, and assigns a designator to it.
* clear_seq(nameseg n); an O-function that clears a designated sequence.
* append(nameseg n; word w); an O-function that adds the word w to the end of the sequence.
* swap_seq(nameseg n; INTEGER i, j); an O-function that causes the words in positions i and j to be exchanged.

The specification of sequences contains three paragraphs. The FUNCTIONS paragraph contains the details of the specification for each function. The DEFINITIONS paragraph contains the definitions of local functions. The TYPES paragraph declares types that are to be referred to in the specification.

The TYPES paragraph must contain the declaration of the designator type introduced in this module. Thus we declare nameseg as the type whose values are the string sequences of interest. Other types, e.g. subtypes or aggregate types, can be declared here. In the sequences specification we declare the aggregate subtype "word". Note that the definition of a word.

the set of all character vectors whose length is positive,

underscores the notion of a type as a set of values. No upper limit on the length of a word is imposed here. In the specification of the

individual functions, we will confront the (inevitable) problem of handling physical storage limitations.

The next module paragraph is the DEFINITIONS paragraph. A function definition is an expression, of declared type, in terms of the V-functions, parameters, or other defined functions of the module. A definition can have arguments or not as required. Thus, the general form of a definition is

typespec defname(typespec1 arg1, ...) IS body

Now let us consider the function specifications in turn.

Stringstate

Stringstate is a hidden V-function that returns the word w at position j in the designated sequence n. As the only non-derived V-function, stringstate captures the "state" of each sequence in the system.

The expression in the INITIALLY section,

w = ?

is shorthand for

initially, for all sequences the value of all positions is ?.

String

String is the visible derived V-function that returns the word w at position j in the designated sequence n. Its derivation is merely the hidden V-function stringstate.

A single exception corresponds to no word being present at position j. The reader might question the absence of any exception condition corresponding to the formal argument n. What if a user invokes string(nn, j) with some designator nn that is not an existing nameseg, possibly being of a different type? It would be necessary to define such an exception only in a context where such a circumstance is expected and must be guarded against. For many types, intended for use in a strictly typed context, such checks would be regarded as unnecessary.

Seqlen

Seqlen is a derived visible V-function that returns the current length of sequence n. The derivation (returned value) is expressed as

> consider an integer set that contains all of the integer positions that store a word whose value is not ?; the returned value is the cardinality of this set.

It is emphasized that this is a specification for determining the number of words in a sequence. It is not an implementation, which would likely be carried out using a memory cell to hold the current sequence length.

Create_Seq

Create_seq is an OV-function that creates a new sequence, initializes it, and assigns a designator to it. To express, as an effect, the generation of a never previously generated nameseg designator we use the notation

NEW(nameseg).

NEW is a predefined function in SPECIAL, that requires an argument of type DESIGNATOR. As part of the underlying semantics of NEW, it never returns "?".

One final note about the specification of create_seq concerns the apparent absence of any effect to express the initialization of a newly created sequence. Such an expression is not needed here since the initial value of stringstate(n, j) is ?, which is precisely what is desired of a sequence after it is created. Thus, the act of creating a sequence is to make a nameseg designator n available so that words can be appended to n, swapped and subsequently read out.

Clear_seq

Clear_seq is an O-function that clears a designated sequence. We express this effect by indicating that the value in all positions of the sequence is to be ?. This specification illustrates how a desirable concise specification can appear to be an over-specification; positions

that were previously ? are re-specified to be ?. An equivalent, but less desirable specification is

```
FORALL INTEGER j INSET {1...seqlen(n)}: 'stringstate(n, j) = ?
```

indicating that all positions in the sequence that previously stored defined words, will have value ? after the invocation. The reader should note that in a specification conciseness is desirable, as contrasted with an implementation where efficiency is generally vital.

Append

Append is an O-function that adds the word w to the end of the sequence. As the effect indicates, after an invocation word w will be at position

```
seqlen(n) + 1
```

which is the newly-created end-position of the sequence. This specification illustrates the purposeful omission in the EFFECTS section of V-function positions whose values one left unchanged. The following expressions are implicit:

```
FORALL INTEGER j ~= seqlen(n) + 1:
   'stringstate(n, j) = stringstate(n, j);
FORALL INTEGER j; nameseg n1 ~= n:
   'stringstate(n1, j) = stringstate(n, j)
```

The first expression indicates that all positions of n except seqlen(n) + 1 are left unchanged, and the second that all positions of all other sequences are left unchanged.

Swap_seq

Swap_seq is an O-function that causes the words in positions i and j to be exchanged. Based on the above discussion the specification should be self-explanatory. Note that no order of operation is implied in the EFFECTS section. After an invocation of swap_seq both expressions will be TRUE. There is no intermediate state.

The Specification of the Module sequences

```
MODULE sequences
          $( maintains an unspecified number of variable length
           sequences of character strings (words) , each string of
           variable length. For reading, words can be randomly
           accessed. New words can be inserted at the end of a
           sequence. Words can be exchanged)


    TYPES

nameseq: DESIGNATOR; $( names of sequences)
word: { VECTOR_OF CHAR vc ¦ LENGTH(vc) > 0 };


    DEFINITIONS

BOOLEAN no_word(nameseq n; INTEGER j)
   IS NOT j INSET { 1 .. seqlen(n) };


    FUNCTIONS

VFUN stringstate(nameseq n; INTEGER j) -> word w;

    HIDDEN;
    INITIALLY
        w = ?;


VFUN string(nameseq n; INTEGER j) -> word w;
        $( returns the j-th string in sequence n)
    EXCEPTIONS
       noword : no_word(n, j);
    DERIVATION
        w = stringstate(n,j);

VFUN seqlen(nameseq n) -> INTEGER v;
   $( returns the number of strings in sequence n)
    DERIVATION
       CARDINALITY({ INTEGER j ¦ stringstate(n, j) ~= ? });

OVFUN create_seq() -> nameseq n;
          $( creates a new sequence all words of which are
           undefined. A newly generated designator is returned)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       n = NEW(nameseq);
```

```
OFUN clear_seq(nameseq n); $( clears sequence n)
    EFFECTS
       FORALL INTEGER j: 'stringstate(n, j) = ?;

OFUN append(nameseq n; word w);
   $( appends word w to the end of the sequence n)
    EXCEPTIONS
       RESOURCE_ERROR;
    EFFECTS
       'stringstate(n, seqlen(n) + 1) = w;

OFUN swap_seq(nameseq n; INTEGER i, j);
   $( exchanges words in positions i and j of sequence n)
    EXCEPTIONS
       no_word1 : no_word(n, i);
       no_word2 : no_word(n, j);
    EFFECTS
       'stringstate(n, i) = stringstate(n, j);
       'stringstate(n, j) = stringstate(n, i);

END_MODULE
```

REFERENCES

1. D. E. Bell, L. J. LaPadula, Secure Computer Systems: Mathematical Foundations and Model, MITRE Corp., Bedford, MA (Sept. 1974).

2. K. J. Biba, Integrity Considerations for Secure Computer Systems, MTR-3153 Rev. 1, MITRE Corp., Bedford MA (April 1977).

3. R. J. Feiertag, K. N. Levitt, L. Robinson, Proving Multilevel Security of a System Design, Proc. Sixth ACM Symposium on Operating Systems Principles, Purdue Univ., West Lafayette IN (Nov. 1977).

4. J. K. Millen, Security Kernel Validation in Practice, CACM vol. 19 no. 5, pp. 243-250 (May 1976).

5. K. G. Walter, et al., Initial Structured Specifications for an Uncompromisable Computer Security System, Case Western Reserve University, Cleveland OH (July 1975).

6. E. W. Dijkstra, "Notes on Structured Programming," in Structured Programming, C. A. R. Hoare, ed. (Academic Press,New York, 1972).

7. D. L. Parnas, "A Technique for Software Module Specification with Examples," Comm. ACM, Vol. 15, No. 5, pp. 330-336 (May 1972).

8. C. A. R. Hoare, "Proof of Correctness of Data Representations," Acta Informatica, Vol. 1, pp. 271-281 (1972).

9. R. W. Floyd, "Assigning Meanings to Programs," Mathematical Aspects of Computer Science, Vol. 19, J. T. Schwartz, ed., pp. 19-32 (American Mathematica Society, Providence, Rhode Island,(1967).

10. L. Robinson and K. N. Levitt, "Proof Techniques for Hierarchically Structured Programs," Comm.ACM, Vol. 20, No. 4, pp. 271-283 (April 1977).

11. L. Robinson, "The HDM Handbook," Volume I: Foundations of HDM,Contract N00123-76-C-0195, Naval Ocean Systems Center, San Diego,California, prepared by SRI International, June 1979.

12. B. Silverberg,K. N. Levitt, and L. Robinson,"The HDM Handbook," Volume II: The Languages and Tools of HDM,Contract N00123-76-C-0195, Naval Ocean System Center,San Diego, California, prepared by SRI International, June 1979.

13. K. N. Levitt, L. Robinson, and B. Silverberg,"The HDM Handbook,"

Volume III: A Detailed Example on the Use of HDM, SRI International, prepared under Contract N00123-76-C-0195June 1979.

14. Ford Aerospace & Communications Corp., "Secure Minicomputer Operating System (KSOS), Verification Plan," Contract MDA 903-77-C-0333,March 1978.

15. J. Guttag, E. Horowitz, and D. Musser, "Abstract Data Types and Software Validation," Comm. ACM, Vol. 21, No. 4, pp. 1048-1064 (Dec. 1978).

16. J. H. Wensley, L. Lamport, J. Goldberg,M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak,and C. B. Weinstock, SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control, Proceedings of the IEEE, October 1978, pp. 1240-1255.

END

FILMED

2-83

DTIC